JavaScript, ¡Inspírate!





por Ulises Gascón González

JavaScript, ¡Inspírate!

Ulises Gascón González

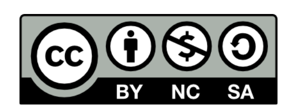
Este libro está a la venta en http://leanpub.com/javascript-inspirate

Esta versión se publicó en 2019-01-11

ISBN 978-84-617-7416-6



Este es un libro de Leanpub. Leanpub anima a los autores y publicadoras con el proceso de publicación. Lean Publishing es el acto de publicar un libro en progreso usando herramientas sencillas y muchas iteraciones para obtener feedback del lector hasta conseguir tener el libro adecuado.



¡Twitea sobre el libro!

Por favor ayuda a Ulises Gascón González hablando sobre el libro en Twitter!

El tweet sugerido para este libro es:

Acabo de conseguir el #ebook JavaScript, ¡Inspírate! de @kom_256 en https://goo.gl/Z2L56H #javascriptInspirate

El hashtag sugerido para este libro es #javascriptInspirate.

Descubre lo que otra gente está diciendo sobre el libro haciendo click en este enlace para buscar el hashtag en Twitter:

#javascriptInspirate

Most good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program.

Linus Torvalds

Agradecimientos

Muchas gracias a todos aquellos que me han asesorado, ayudado y opinado sobre los drafts de este trabajo.

Tampoco hubiera sido posible sin ayuda de Open Source Weekends, especialmente a Ignacio Villanueva, que diseñó la increíble portada, además de participar en la revisión junto con José Gallego, Nuria Sanz, Carlos Hernandez y Roberto Moriyón.

Este libro es producto, de un aprendizaje continuo y profundo, añadido a un intenso y constante esfuerzo personal, que proyecta todo lo que he aprendido siendo profesor, en las empresas que he trabajado y las que he asesorado. Así como las numerosas aportaciones de mis colegas de profesión y las de mis propios alumnos.

Estos agradecimientos no estarían completos, si no reconociera la intensa labor que han realizado los grandes desarrolladores que nos han precedido, que a día a día se esforzaron en levantar una comunidad libre y una filosofía que me inspira para seguir siendo un "artesano del código".

Casi todo este libro ha sido escrito a -medio camino- entre Tetuán Valley y la cafetería del Campus de Google en Madrid.

Agradecimientos a la comunidad

Por ayudarme a mejorar el libro una vez publicado reportando errores y proponiendo mejoras desde nuestro repositorio:

- *Jorge Muñoz Rodenas*
- *Carlos Bravo*
- *Yoel Macia Delgado*
- *Andrés* `(@7v1r)`
- *Theba Gómez*
- *Álvaro Ramírez*
- *Fausto Gutiérrez*
- *Ana Enríquez*
- *Antonio Garcia Marin*
- *Faustogut* `(@faustogut)`
- *Ana* `(@anakb)`
- *Victor* `(@victorze)`
- *Queli Coto*
- *Juan A. Macías*

En otros idiomas

Este libro originalmente fue escrito en español, pero ha sido traducido por la comunidad a los siguientes idiomas:

- *Gallego. Gracias a Yoel Macia Delgado*

Sobre el autor



Desarrollador Full Stack JavaScript y Python. Especializado en Internet of Things con hardware libre como Arduino y Raspberry Pi.

Colaborador activo en la comunidad de desarrollo de Software Libre, co-organiza Open Source Weekends.

En la actualidad trabaja como consultor independiente y es profesor, impartiendo formación sobre JavaScript, Node.js, Python e Internet of Things, entre otros.

Redes Sociales: Github | Twitter | Linkedin | Blog

Sobre el libro

version: v1.0.2

Índice general

Parte I - Hablemos de JavaScript

Introducción

Sobre este libro

He tenido grandes maestros, que me han enseñado mucho y he leído muchos libros sobre este mundillo… pero nunca he logrado dar con el libro que solucionará todas las dudas o me diera una base sólida sobre la que crecer en un lenguaje tan fascinante y estigmatizado como es JavaScript.

Por eso, este libro está escrito como me hubiera gustado que fueran aquellos que yo leía cuando empezaba a programar en JavaScript.

Este libro que tienes ahora mismo ante tus ojos, es el resultado de mi esfuerzo personal, con el objeto de crear un instrumento sencillo y simple que muestre al lector la base de un lenguaje tan peculiar y extendido como JavaScript.

Mi idea, es empezar desde cero y hacer un viaje juntos desde los abismos profundos de la duda, pasando por el pseudocódigo hasta llegar al maravilloso mundo de las *llamadas Ajax*, haciendo muchas paradas en el camino, en las que aprenderemos todo lo que necesitas para empezar tu aventura como desarrollador *Front-end* con sólidas bases en JavaScript.

Esto va de ser una comunidad

Este libro está licenciado bajo *Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)*, lo que permite compartir, distribuir, adaptarlo siempre y cuando no lo hagas con fines comerciales, atribuyas el crédito al autor y compartas las modificaciones bajo la misma licencia.

Todo el código fuente, está en *este repositorio de GitHub*, desde este lugar espero poder ayudaros con las dudas que os puedan ir surgiendo, y también cuento con vuestras -siempre interesantes- aportaciones y sugerencias a través de *estos issues*

Este libro es algo vivo y por ello estará evolucionando constantemente. Recuerda que tu puedes formar parte de todo esto, colaborando.

Existen muchas formas de colaborar:

- Avisando de los errores y de las posibles erratas que pudieses encontrar en el código.
- Mejorando los ejemplos.
- Traduciendo este libro a otros idiomas para que llegue a más lectores.
- Compartiendo tus sensaciones en twitter con el hashtag *#JavascriptInspirate*.

Encontrarás más información en el archivo *contributing.md*, dentro de nuestro *repositorio*.

¿Qué necesito saber antes de empezar?

No es necesario tener experiencia programando, pero si que es importante tener claro el contexto de cómo se hacen páginas web estáticas, ya que en la parte III nos centraremos en la manipulación dinámica del *DOM* y las *llamadas AJAX*.

No me extenderé mucho acerca de *HTML* y *CSS*, por lo que espero que ya tengáis cierto conocimiento adquirido. Aunque vuestro nivel de conocimientos y experiencia sea relativamente bajo, hemos procurado evitar que esto sea un freno a tu aprendizaje en JavaScript.

Si tienes experiencia en otros lenguajes de programación, jugarás con ventaja al principio, pero no te confies…

Si ya has trabajado con *JQuery* hablaremos de como hacer todo lo que haces con JavaScript en los últimos capítulos.

¿Qué aprenderemos?

- Pseudocódigo.
- A pensar como un programador.
- Trabajar fluidamente con funciones.
- Dominar las bases de JavaScript (estructuras del lenguaje).
- Ajax y sus peticiones.
- Manipulación dinámica del DOM.

¿Qué no aprenderemos?

JavaScript es un universo inmenso que va creciendo a pasos agigantados con expansiones. Recuerda que la idea es sentar las bases del lenguaje, por lo que no veremos cosas como:

Bases de datos
Ya sean *SQL* o *NoSQL*, es algo que no pretendemos tratar en este libro.

Paradigmas *POO*, *Funcional*, etc…)
Hablaremos de pasada de algunos conceptos interesantes de los diversos paradigmas que ofrece JavaScript, pero no iremos más allá.

HTML5 APIs
Aunque hablaremos de pasada sobre cosas tan potentes como *LocalStorage*, queda muy lejos de las bases de JavaScript sobre las que queremos centrarnos.

Regex (*Expresiones Regulares*)
Es algo muy útil en el día a día, y será mencionado, pero solo por encima para entender como enfrentarnos a ello como desarrolladores noveles.

Patrones de diseño (*Decorator*, *Façade*…)
Son muchos *los patrones* que podemos aplicar a JavaScript para que nuestro código adquiera “super poderes”, pero esto también queda muy lejos de nuestros objetivos.

Testing
Uno de los pilares del buen código, es hacer testing o seguir filosofías como *TDD* o *TBD*. Pero para alguien que empieza en este mundillo, creo que sería un freno a tu aprendizaje.

Git* y *Github
Otra de las cosas sin las que uno no puede considerarse un desarrollador de verdad, si no sabe manejarse con fluidez en ello.

Frameworks *Angular*, *React*, *Polymer*, *Backbone*, etc…)
La fiebre MV de JavaScript* quedará fuera de este libro por motivos evidentes pero recuerda que aqui utilizaremos el *framework* de JavaScript más popular y veterano, *Vanilla-js*.

Librerías *Jquery*, *Mathjs*, *Momments.js*…)
La idea de este libro es trabajar sobre JavaScript -vieja escuela-, al estilo de *Douglas Crockford*. Por eso no trabajaremos con *JQuery* directamente, ni con ninguna otra librería, aunque si recomendaremos otras muchas librerías.

ECMAScript 6, ECMAScript 7
Aunque hablaremos de ciertas cosas que ha traído consigo *ECMAScript 6*, sin embargo nos centraremos en el clásico *ECMAScript 5.1* (vigente desde 2011) con una *compatibilidad nativa* envidiable.

Node.js
Me encanta *Node.js*, por tener todo el potencial de JavaScript, sin las restricciones del navegador… bucear en un *mundo asíncrono* de posibilidades infinitas donde puedes crear desde *servidores Http*, hasta cosas tan alucinantes como *robots*…

Node.js da para una colección de libros por si mismo, y por ello queda muy lejos de nuestros objetivos.

***Gulp*, *Grunt*, *Broccoli*…**
Aunque la automatización y la gestión de tareas es otra de las cosas que más me apasionan, queda también lejos de los objetivos que nos hemos marcado. ¡Lo siento!…

De paso también me gustaría desmitificar algunas cosas:

Matemáticas
Programar es hacer magia con máquinas, las matemáticas son opcionales…

Matemáticas Avanzadas, Estadística, etc…
Para dominar los conocimientos y las técnicas de JavaScript, solo necesitas… saber sumar, restar, dividir y multiplicar… ¡solo eso!

Física nuclear
Un poco de humor.

Inteligencia Artificial
Esto queda un poco lejos aún, pero sigue trabajando… ¡Padawan!

Importante

Humor
Si no te ríes en algún momento mientras lees este libro…. claramente no es para ti y no he cumplido el reto de intentar enseñarte cosas alucinantes con un poco de humor :-D

Cero coste, 0$
Este libro es completamente gratuito, con un doble objetivo, que nadie se quede sin aprender por no tener recursos y además para contribuir de este modo a la comunidad del *software libre*.

Aunque el libro es gratuito, *Leanpub* permite comprar este libro al precio que tu consideres. Todos los beneficios generados serán donados integramente a *Code Club* desde la propia plataforma de *Leanpub*. El autor no percibirá nada de esas donaciones.

Adaptable
En principio este libro está disponible en tres formatos básicos *pdf*, *mobi*, *epub* y su código fuente está en *Markua*. Pero partiendo del repositorio en GitHub puede ser convertido a más formatos.

Este libro está vivo
Con el tiempo irá creciendo y mejorándose. Mantente al día desde su *página oficial*

Un mar de recursos
La mayoría de los links del libro, os dirigen a blogs técnicos y a la documentación oficial de *Mozilla Developer Network*. Si un concepto resulta difícil de entender o deseas profundizar más… estos links pueden ser un buen punto de partida.

Facilidades
La mayor parte del código está creado con la nomenclatura en español, para facilitar la comprensión y lectura por parte de nuevos desarrolladores. No obstante, recuerda que esto no está considerado una buena practica en entornos profesionales.

¡A tu ritmo!
A lo largo del libro observaréis que hay numerosos ejemplos explicativos. Sin embargo no encontraréis ejercicios, para así hacer más ágil su lectura y para que os animéis a venir a una de las muchas reuniones de la comunidad (*Open Source Weekends*, *Node Girls Madrid*…) y practicar con proyectos de verdad y con el tiempo lleguéis a proponer vuestros propios proyectos dentro de la comunidad.

Convenciones utilizadas en este libro

Ejecuta el código

Todos los ejemplos de código que veras en el libro tienen significado por si mismos y pueden ser ejecutados en la consola del navegador sin problema alguno, es más, os lo recomendamos encarecidamente.

Iconos utilizados en este libro



Cuando veas este icono sabrás que tendras un lugar desde el que ampliar conocimientos y profundizar un poco más en los conceptos del capítulo.



Cuando veas este icono sabrás que debes prestar mucha atención, ya que se trata de un concepto clave para entender el capítulo en el que te encuentras.



Cuando veas este icono sabrás que se trata de un consejo que puedes seguir o no…



Cuando veas este icono sabrás que estamos hablando de cosas relacionadas con la comunidad de desarrolladores.



Cuando veas este icono sabrás que los enlaces te llevarán a *web.archive.org*, la máquina del tiempo de Internet.



Cuando veas este icono sabrás que ha llegado el momento de preparar un buen café y dedicar un tiempo a leer con calma los links y ver algunos vídeos también.

Capítulo 1 - JavaScript de hace 10 minutos



Batallitas del abuelo

En este capítulo hablaré de la historía de nuestra industria y como hemos llegado hasta donde estamos. Algunos lectores ya estareís familiarizados con esto y no necesitaís repasar nuestra historia. En tal caso… simplemente pasad al siguiente capítulo.

Voy a contarte una historia que narra el sacrificio y la lucha diaria de un grupo inmenso de personas por todo el mundo que creyeron, que una idea tan genial como interconectar ordenadores podría llegar a ser la mejor forma de compartir el conocimiento de la humanidad, convirtiendo Internet en una maravillosa y muy valiosa herramienta para la humanidad.

Para llegar hasta donde estamos hoy… mucha *sangre de unicornio* se ha derramado.

Y ahora nos pondremos serios para hablar de nuestra industria.

La historia de nuestra industria



Lecturas recomendadas:

- *History of the Internet by Melih Bilgil*
- *What is the Internet? by Code.org*

La Evolución de la web desarrollada por el *equipo de Google Chrome* es un portal que nos permite visualizar los hitos más importantes en la historia de la web desde 1991 hasta 2013.

1991 - 1993

Los primeros años de la web pasan en mucha calma. Se crea *HTTP* y HTML1 dando el pistoletazo de salida para que empezara el mundo web. El primer navegador web gráfico, *Mosaic*.

1994 - 1995

Surgen los tres navegadores principales de la época; *Netscape Navigator*, *Opera* e *Internet Explorer*. Se oficializa HTML2 y rápidamente es sustituido por HTML3.

Por primera vez se pueden usar *cookies*, pero aún estábamos muy lejos de tener que poner *disclaimers*, claro que por aquel entonces, internet era mucho más salvaje y menos regulado. En un intento firme para hacer la web más segura, y permitir cosas tan futuristas por aquel entonces como el comercio electrónico, se implementa la *capa SSL* y asi se crea el *HTTPS*, *IMAPS*, *POP3S*, *SMTPS*, *FTPS*, etc…

Corría el año 1995, cuando *Brendan Eich* creo Mocha, a petición de *Netscape* con la idea de tener algún lenguaje de programación disponible en la web (lado de cliente). Luego se renombraría como LiveScript, para finalmente ser JavaScript. Una estrategia de marketing dudosa, que sigue hoy en día sembrando dudas planteando que *Java* y JavaScript son cercanos o iguales, lo que es *completamente falso*.

Además a lo largo de ese año, comienza una escalada de problemas para todos los desarrolladores del mundo, conocido como *"Browser Wars"* que hasta hoy en día, seguimos sufriendo sus efectos.

Básicamente, cada navegador decide hacer las implementaciones de los distintos componentes que vendrán en los años venideros de manera diferente, lo que obligaba a los *webmasters* a tener que hacer su código compatible para todos los navegadores al mismo tiempo, replicando esfuerzos y entorpeciendo el trabajo diario. Si a esto le sumamos que por aquel entonces los navegadores no se autoactualizaban, llegamos así a una situación realmente compleja y atípica que podría haberse evitado… pero por aquel entonces Internet era tan solo un producto novedoso con el que no se sabía muy bien como se monetizaría nada.

Para intentar recordar un poco como era navegar entonces, algunos nostálgicos recordarán la usabilidad de la web en aquella época, con contadores de visitas, gifs animados, fotografías pixeladas y mezclas de color únicas. Era la infancia de la web, y aún se puede recordar en sitios como *The world's Worst Website Ever*.



The world's Worst Website Ever - Logo

Dejamos esta época de internet con *39,14 Millones* de usuarios.

1996 - 1998

Arrancó esta época con un número inicial de usuarios de 100 millones, para terminar con 183,91 millones.

Internet Explorer introduce una nueva etiqueta html *iframe*, que permite la carga de una página web dentro de otra.

En esta época, todavía se maquetaban las paginas web *dentro de tablas*. Conceptos como *hojas de estilos* estaban muy lejos de ser viables. Van surgiendo tecnologías claves para suplir carencias básicas de *interacción*, *interconectividad* y *visualización*.

En esta época surgen tecnologías como los *Applets de Java*, *Flash*, etc…

XML se convierte en el formato de intercambio de datos más extendido.

En cuanto a HTML, sigue evolucionando, pasando rápidamente a la version 3.1 y 4, permitiendo la separación de la parte visual con CSS2. Por primera vez contenido y disposición gráfica de la información se separan.

Empieza la era dorada de Flash. Por el momento *CSS* solo es un chiste comparado con las capacidades de Flash.

Mosaic muere en su versión 2.1, Internet empieza a ser un mercado potencial. Se crean los primitivos sitios web de Google, Yahoo!, Altavista y Amazon.



Web Archive:

- *Google Prototipo en 1998*
- *Google Beta en 1998*
- *Amazon en 1998*

1999 - 2007

Fueron tiempos locos… con el cambio de siglo, Internet se consolido creando la famosa *“Burbuja puntocom” que empezó a cocinarse entorno a 1997, pero que termino de explotar en 2001-2002*.

Sobrevivimos *al efecto 2000*, con 305,09 millones de usuarios.

Internet Explorer introduce *ActiveX* en la web. Se introducen los primero borradores de Ajax. En 2006 se introduce en los navegadores *XMLHTTPRequest2* y empieza la navegación asíncrona, lo que permite crear webs realmente dinámicas, que no necesitan una gran renderización en el lado del servidor.

Esto hace posible la mejora en el desarrollo de las primeras *plataformas de redes sociales (Web 2.0)*. Poco a poco en esta época, el peso de JavaScript va creciendo, ya que gran parte del procesamiento pronto se hará en el lado del cliente.

En la parte más visual se empieza a introducir de manera experimental conceptos como *SVG* y *Canvas*.

Aparecen en escena Facebook, Twitter, Linkedin, Youtube, etc… creando el embrión de lo que luego serían otras redes sociales.

Aparece *JQuery*, una librería para JavaScript que cambió para siempre la forma de interactuar entre JavaScript y HTML/CSS. En los años venideros, muchos maquetadores utilizarán esta librería como puerta de entrada a JavaScript.

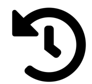
La época dorada del *PHP (pre-Wordpress)*, como motor de innovación para la web con proyectos como *PHPBB*, llegará a su fin.

Con la llegada de *Wordpress* (2003), *Joomla* (2005), etc… la web sufrirá en años sucesivos un cambio drástico.

Los proyectos se dividirán en aquellos que puedan ser realizados con un *CMS* (principalmente en PHP) y otros que necesitarán de un trabajo a medida.

El mítico Netscape se despide para siempre con la versión 9 y nace Firefox.

Termina 2007 con 1.500 millones de usuarios en Internet.



Web Archive:

- *Amazon en 2000*
- *Google en 2000*
- *Google en 2005*
- *Facebook en 2005 (AboutFace)*
- *Facebook en 2005*
- *Amazon en 2005*
- *Youtube en 2005*

2008 en adelante…

Empieza una nueva era… lo nativo va poco a poco desterrando las soluciones arcaicas y obsoletas como Applets, Flash, ActiveX…

HTML5 y CSS3 dan el empujón que hacía falta para que la web despegara. *Chrome* nace justo aprovechando estos cambios de paradigma.

Todo aquello que podia hacerse con Flash, ahora podrá hacerse mejor y más rápido de manera nativa juntando las tres fuerzas más imponentes de la web, CSS3, HTML5 y JavaScript.

Se abre el desarrollo hacia una web mucho más *semántica* con *filosofías responsive*, y empieza la era del "Internet en la palma de la mano" con los *smartphones*.

La popularidad y capacidad de la nueva Internet ocasionarán un cambio de paradigma que trastocará todo. Poco a poco, las aplicaciones de escritorio serán sustituidas por aplicaciones web, ocasionando una migración masiva de desarrolladores hacia el trio de la web (HTML, CSS y JavaScript).

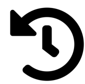
A lo largo de los años siguientes, serán muchas las empresas que decidan dar soporte a JavaScript en sus productos para asegurarse acceder a una masa crítica de programadores, este es el caso de los *scripts para Photoshop*, *Unity*, etc…

También veremos que la complejidad de conseguir desarrolladores de APPs Nativas (*Java*, *Swift* y *Objetive-c*) forzará a que lleguen los sistemas híbridos, donde JavaScript, HTML y CSS serán todo el conocimiento necesario para entrar en el mundo de los smartphones, gracias a soluciones como *PhoneGap*.

La fiebre por extender JavaScript, irá mucho más allá de lo imaginado y encontrará en *Node.js* la pieza que faltaba para encajar en ciertos entornos como las *aplicaciones de escritorio*, o la gestión de redes.

Todo ello, permite que JavaScript deje de ser un lenguaje exclusivo de la web para ir mucho más allá, adentrándose incluso en el desarrollo del *Internet of Things (IoT)* y la *Robótica* con librerías como *Cyclon.js*, *J5*, etc…

Internet de *alta velocidad* y la *fibra óptica* supusieron una gran mejora en las comunicaciones, lo que ayudó a crear una Internet mucho más rica en contenido. En 2011 ya se manejaban 27.483 PB/mes, frente a los 10.000 PB/mes de 2007. La logística y las telecomunicaciones jugarán un papel importantisimo en el futuro desarrollo de nuestro planeta.



Web Archive:

- *Google en 2010*
- *Google en 2015*
- *Amazon en 2010*
- *Amazon en 2015*
- *Youtube en 2010*
- *Youtube en 2015*
- *Facebook en 2010*
- *Facebook en 2015*

El largo camino del Developer

Este esquemático *mapa mental*, puede ayudaros a visualizar lo que está ocurriendo hoy en día en la web, y comprender como los profesionales de la industria, hemos ido migrando de nombres y funciones hasta donde estamos ahora mismo, que no es definitivo y lógicamente seguirá evolucionando.

Una historia de evolución

Muy lejos quedan los tiempos en los que ser *Webmaster* era suficiente para gestionar todo lo que tenia que ver con la web, desde programación a gestión de los usuarios, pasando por el primitivo Marketing de la época.

La web fue evolucionando rápidamente a nivel de desarrollo, aquel *webmaster* multiusos, paso a dividirse en dos grandes y muy diferenciados roles:

Diseño

Encargado de hacer los diseños básicos con algún programa de la *suit de Adobe* o similar.

En ocasiones también se encargaba de la parte de Flash, para crear animaciones y transiciones.

Programación

Realizaba todas las tareas de desarrollo: JavaScript, PHP, Bases de datos, formularios, *hosting*, etc… Las webs de entonces no eran muy complejas, gran parte de la lógica se hacía en el servidor y el verdadero reto era lograr los objetivos con la tecnología de la época.

Aquellos eran los viejos tiempos… era como construir los primeros aviones. Materiales como la tela y madera servían para construir todo el entramado del fuselaje, un motor potente, unos controles sencillos y ya estabas volando.

A medida que la web fue evolucionando, su complejidad también creció exponencialmente, y como consecuencia inmediata, la programación se dividió en dos grandes áreas.

Diseñador/Maquetador

Antes con el diseño era suficiente. Pero poco a poco, el diseñador va asumiendo competencias básicas, para descargar de trabajo meramente estático a los programadores, que se centrarían en hacer la parte menos visual.

Ahora el diseñador tomará la iniciativa y convertirá los diseños en HTML y CSS.

Front-End developer

Algunos desarrolladores, deciden que asumirán las funciones de interacción del lado del cliente (JavaScript) y dejando el servidor. En muchas ocasiones, el diseño quedará fuera de sus competencias.

A medida que JavaScript dejó de usarse solo para interpretación y empezó a ganar fuerza frente al renderizado total de datos en el servidor, se migrará de JQuery hacia JavaScript nativo y a las librerías MV* como Angular, Backbone, Ember…

Back-End developer

El desarrollo en el servidor también sufrirá muchos cambios. Poco a poco, se migrará de proyectos web que basan la mayor parte de su programación en el renderizado de HTML, CSS y JavaScript, desde el servidor a la creación de *APIs*, donde prima el aislamiento del procesamiento de la información.

En cuanto a lenguajes de programación, poco a poco se irá viendo que más y más sistemas y lenguajes se esforzarán por entrar en el mundo de internet con nuevas librerías y arquitecturas. *Apache* logra hacer de puente para muchos, pero a la larga surgirán alternativas.

Mientras tanto, muchas iniciativas asaltan la web con ideas innovadoras como *desarrollo ágil*, *eXtreme Programming (XP)*.

El *software libre* acabó convirtiéndose en el standard dentro del sector gracias al *mítico LAMP*.

Las bases de datos también evolucionarán y tendremos que convivir con dos maneras de entender el almacenaje, *SQL* y *NoSQL*. Lo que supondrá una dificultad añadida para el desarrollo de nuevos proyectos.

Otras áreas como la *inteligencia artificial* y *Big data* irán poco a poco demandando su hueco, por lo que dentro del backend iremos viendo nuevas áreas de especialización.

Full Stack Developer
Surge una nueva clase de desarrolladores, que por avatares del destino no se encasillan en el *back* o en el *front*.

Y serán capaces de adentrarse en ambos mundos y suplir las necesidades de los equipos en estos dos frentes. Cada Full Stack Developer será diferente, cada uno será especialista en unas areas, y en otras pasará de largo.

Mantenerse al día en la industria es cada vez más difícil. En cuestión de unos pocos años pasamos de construir aquellos aviones básicos con tela y madera a desarrollar cohetes capaces de *surcar una galaxia*.

Nuestra industria se vio totalmente superada. El crecimiento de Internet no era ni de lejos acorde a la cantidad de profesiones capaces de suplir esta demanda. Esto creó un *efecto llamada* que desató la llegada de muchos nuevos desarrolladores, que no sienten esta profesión con la pasión de aquellos que empezamos a construir un Internet mejor, nosotros realmente creíamos en lo que hacíamos y como lo hacíamos.

La asimilación de programadores obsoletos de otras áreas y ajenos a la industria, condicionará mucho la manera de trabajar, especialmente con JavaScript.

Básicamente podría decirse que esta industria se divide entre artesanos y mercenarios.

Una realidad caótica.

A medida que esto crece, las barreras para entrar en la industria o cambiar de rol dentro de ella, aumentan.

Para hacernos una ligera idea de lo que necesitamos saber en las diversas áreas, os dejo este *esquema*.

Cosas que todos deberíamos saber

- Manejo de Git/Github.
- Entender como funciona HTTP y las APIs.
- Manejo básico de terminal/consola.
- Trabajar con FTP.
- Manejo de SSH.

¿Qué se espera de un Front-End Junior?

- Conocimiento bueno de HTML5.
- Conocimiento bueno de CSS3.
- Bases de testing.
- Llamadas AJAX.
- Trabajo con librerías como JQuery.
- Conocimientos básicos de alguna librería MV*.

¿Qué se espera de un Front-End Senior?

- Lo mismo que de un Junior, pero en mayor profundidad y con experiencia.
- Gestión de tareas *Gulp*, *Grunt*, etc…).
- *Preprocesadores CSS*.
- Trabajo con *Flexbox*.
- Dominio del desarrollo Responsive.
- Gestión de dependencias *RequireJS*, *Webpack*, *Browsefy*, etc..).
- Dominio/manejo fluido de un framework MV*.
- Conocimientos de Node.js.
- Conocimientos muy sólidos de JavaScript *Patrones de diseño*, ES6, etc..).
- Conocimientos avanzados sobre HTML5 APIs.
- Buenas prácticas (documentación, *refactorización*, etc…).

¿Qué se espera de un Back-End Junior?

- Conocimientos sólidos de programación (Algorítmia, lógica, etc…).
- Saber trabajar con bases de datos relacionales y/o no relacionales.
- Bases de una buena arquitectura (modularidad, *microservicios*, etc…).
- Bases de nuevas arquitecturas *NoBackend*, *Serverless*, *WebSockets*, etc…).
- Desarrollo de aplicaciones seguras.
- Conocimientos de protocolos como *OAuth*, HTTP (server side), etc..
- Al menos un lenguaje de programación con un buen nivel.
- Trabajar con fluidez con un motor de plantillas.
- Bases de HTML5 y CSS3.

¿Qué se espera de un Back-End Senior?

- Lo mismo que de un Junior, pero en mayor profundidad y con experiencia.
- Despliegue en Servidores y servicios en la nube.
- Al menos dos lenguajes de programación de Backend.
- Buen nivel de JavaScript.
- Bases de Node.js para la automatización *Yeoman*, *Gulp*, etc..).
- Bases de la Integración continua *Jenkins*, *TravisCI*, etc…).
- Experiencia con varias bases de datos diferentes.
- Gestión de cacheo *Nginx*, etc…).
- Creación de APIs.
- Integración con otras plataformas y servicios.
- *Shell Scripting*.

En el centro del huracán: JavaScript



Lecturas recomendadas:

- *The Deep Roots of Javascript Fatigue by Calvin French-Owen*
- *El estado del desarrollo Front-End en 2015 por Ashley Nolan*
- *How it feels to learn JavaScript in 2016 by Jose Aguinaga*
- *State of js*
- *How to avoid JavaScript fatigue and sleep well at night by Josh Mock*
- *The Magpie Developer*
- *The Sad State of Web Development*
- *A response to The Sad State of Web Development — Its not about Javascript really*
- *How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript*
- *Is left-pad Indicative of a Fragile JavaScript Ecosystem?*
- *Overcoming JavaScript Fatigue*
- *¿Y si el software Open Source desapareciera?*

Para los desarrolladores de JavaScript, todo ha sido complicado, ya que el crecimiento exponencial de las necesidades de todo Internet ha pasado por este lenguaje.

Al ser el único lenguaje que se puede ejecutar en el navegador, muchos desarrolladores han tenido que pasar por el aro y aprenderlo.

Otros muchos, han tenido que crear librerías y frameworks para intentar hacerlo más sencillo y ágil.

Estos y más factores, han hecho que lleguemos a un punto de no retorno que oficialmente se conoce como *JavaScript Fatigue*.

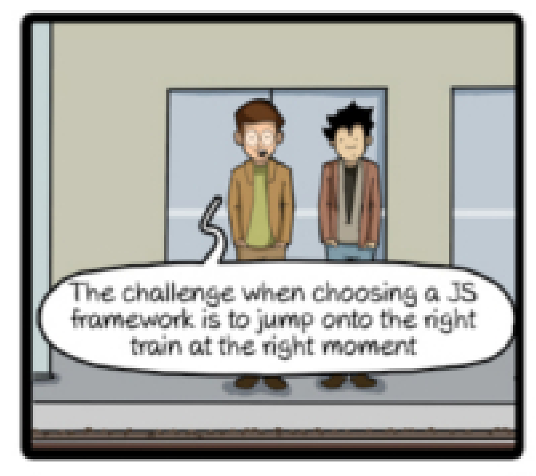










Imagen de CommitStrip.com

Durante estos años *JavaScript se hace adulto* de la mano de *Ryan Dahl* cuando se libera *Nodejs*.

Node.js supone un antes y un después en toda la industria. No solo para los desarrolladores de JavaScript, ya que Node.js no es únicamente un entorno para desarrollar *servidores de internet clásicos (Http)*. Ya que su fuerte está en los nuevos paradigmas que maneja.

Cosas como la asincronía, la orientación a eventos y el paralelismo… lo convierten rápidamente en uno de los entornos de desarrollo más rápido.

Que fuera multiplataforma, se basará en el *motor V8 de Google* e incluyera un gestor de paquetes tan evolucionado como *NPM*, hacen de Node.js una herramienta ideal para hacer cosas tan diversas como aplicaciones de red, automatización de tareas, etc…



IO.js

Y no tardaron mucho en llegar los *#jsDramas* cuando la comunidad de Nodejs decide separarse en dos, *IO .JS* y Nodejs.

Creando una brecha que muchos consideraron insalvable. Al final Node.js reabsorbe a IO.js y acepta todos los cambios y la evolución que se deseaba originalmente.

Mas información…

A su vez, Node.js crea un peculiar efecto de absorción y fusionamiento de roles entre el frontEnd y backEnd, ya que ahora no existe una barrera de lenguaje real entre el back y el front. Solo es necesario aprender Node.js para entrar en el mundo del back.

Como consecuencia se crean nuevos developers de JavaScript, a los que cada vez más, se les exige conocer y manejarse con Node.js.

Revolución… ¡Revolución!



Lecturas recomendadas:

- *History of Gnu, Linux, Free and Open Source Software (Revolution OS)*
- *Platform as a service (PaaS)*
- *Manifiesto por el Desarrollo Ágil del Software*

A lo largo de estos años, muchas cosas han cambiado en el mundo del desarrollo del software.

Para empezar se crearon nuevas maneras de entender el entorno de desarrollo. Ya no solo basta con crear nuestro código y subirlo al servidor *vía FTP* sin más.

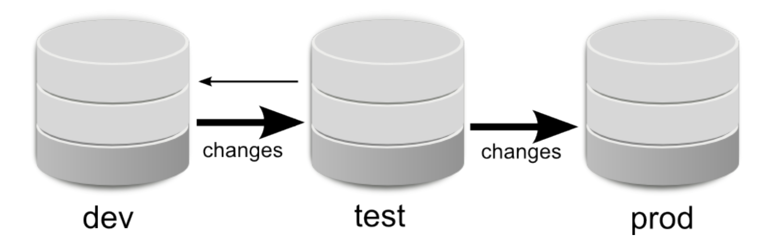


Imagen de z2-environment

Ahora es importante seguir una filosofía de desarrollo más compleja, donde debemos usar un *control de versiones* como *Git*.

Nuestro entorno local ya no se conectará directamente al servidor de producción para subir los cambios en los proyectos, pasaremos primero por el entorno de testing, y una vez sepamos que todo funciona como debería, será subido a producción.

Las subidas a producción, deberán de ser frecuentes, ya no será necesario tener una *release* completa para hacer subidas a producción.

Entramos en la era del desarrollo ágil, la *programación extrema* y la *computación en la nube*. El tener servidores físicos en los sótanos de la empresa, se eliminará a favor de sistemas como *Google Cloud*, *Amazon Web Services*, *Heroku* u otros proveedores.

Todo será mucho más modular y escalable, todo estará mucho más interconectado, el *login social* cambiará la manera de autentificarnos.

Ya no reinventaremos la rueda tan a menudo y dependeremos mucho más de APIs y sistemas de terceros.

El *software libre* dejó de sonar a cosa rara.

La mayor parte de los servidores de internet montarán *distribuciones Linux*. La colaboración entre desarrolladores de manera altruista bajo el sistema del software libre, creará algunos de los lenguajes, librerías y sistemas más sólidos.

Esta idea de libertad, se extiende como la pólvora y da pie a extender estas filosofías a otras áreas como a la cultura con licencias *Creative Commons* o al mundo del hardware con el *Hardware libre*

Al tener un entorno libre del que nutrirse muchos desarrolladores empiezan a innovar. Rápidamente la comunidad les sigue, se van constituyendo nuevos ciclos y formatos de innovación.

Algunas empresas, poco a poco, verán este valor y migrarán de un modelo más tradicional de software privativo a uno más abierto y libre.

Cambiarán muchas cosas, *XML* será destronado y *JSON* será el nuevo rey. Se idearán nuevas maneras de gestionar el tiempo real como *WebSockets*.

Surgirán iniciativas con ideas tan potentes como *Nobackend*, que plantearán nuevos paradigmas, pero sobretodo la innovación y la creatividad empujarán sobre manera -como nunca antes- el mundo del software.

Precisamente la innovación pasará a formar parte del ADN de nuevos tipos de empresas como las *Startups*. El *MVP* y el prototipado serán un concepto *mainstream* al igual que la metodología *Lean Startup*.

Capítulo 2 - Hola Mundo



Fiero veterano

En este capítulo, hablaremos sobre como es la computación y que se espera de un desarrollador. Si ya dominaís algún lenguaje de programación o sencillamente queréis empezar ya mismo con JavaScript, pasad al siguiente capítulo.

Un mundo de máquinas

Seguramente, cuando abras este libro tendrás un objetivo sencillo en mente, aprender JavaScript.

Pero antes de hablar de JavaScript, no es mala idea emplear un poco de tiempo, en aprender algunas cosas de interés sobre el entorno de JavaScript.

JavaScript es un lenguaje más de tantos que existen. Cada lenguaje tiene unas características y unas propiedades especiales que lo hacen más o menos apto para según que objetivos. Eso explica la *existencia de tantísimos lenguajes de programación*, y que los programadores nos repartamos en función de los lenguajes que utilizamos y manejamos.

Los lenguajes de programación y los que utilizamos los humanos no son tan diferentes en cuanto al objetivo principal, que es comunicarnos. En el caso de los lenguajes de los humanos la idea principal es que podamos comunicarnos y transmitir nuestras ideas entre los individuos. Cuando se habla de lenguajes de programación, el objetivo es lograr establecer una comunicación fluida con una máquina/sistema/protocolo/etc…

Al igual que en los lenguajes humanos encontramos muchas similitudes entre si en cuanto a estructuración, los lenguajes de programación suelen compartir una base común en cuanto a estructuras y funcionalidades que toda máquina es capaz de comprender, tales como funciones, bucles, variables, condicionales, etc…

La parte más pura de la computación, *lógica*, *abstracción* y la capacidad de sintetizar una solución para un problema, en *sentencias* que una máquina pueda entender y ejecutar.

Los ordenadores son *máquinas multipropósito* que pueden ser programadas. En función de su programación pueden hacer más o menos cosas y hacerlas de una forma concreta.

Por si mismas, las máquinas no son capaces de razonar y tomar sus propias decisiones, excepto, que hablemos en el ámbito de la inteligencia artificial, donde se busca entre otras cosas lograr máquinas y sistemas que aprendan.

Por lo general, en computación siempre hablaremos de sistemas dentro de sistemas y que han sido programados… es como una pirámide que cobra todo su sentido cuando por fin tenemos cada uno de sus componentes correctamente alineados.

JavaScript es un lenguaje de programación de alto nivel, que se ejecuta dentro de los navegadores, a excepción de Node.js. Esto quiere decir que nuestro código necesita mucho código previo para funcionar.

Por ejemplo, un sistema operativo (Linux, Windows, Mac..) que gestiona programas como el navegador (Firefox, Chrome, IE, etc…) y estos a su vez gestionan cómo se representan los entornos visuales en las pantallas, los sonidos, etc…

Sabiendo todo esto, al final del día nuestro código puede sufrir fluctuaciones a la hora de ejecutarse en función del entorno que estemos usando, por suerte, el navegador gestionará de manera automática gran parte de todo esto, pero aún así podemos sufrir gran cantidad de variaciones debido a que cada navegador "interpreta" nuestro JavaScript lo mejor que puede, aunque no de la misma forma.

Uno de nuestros principales cometidos, será hacer que nuestro código sea capaz de lograr los objetivos que nos proponemos independientemente de la plataforma donde se vaya a ejecutar, aunque a esto ya estaréis familiarizados con el día a día desarrollando con HTML5 y CSS3.

Pensar como un programador



Lecturas recomendadas:

- *Don't learn to code. Learn to think by Yevgeny Brikman*
- *Linus Torvalds: La mente detrás de Linux*
- *What Most Schools Don't Teach by code.org*
- *Computer science is for everyone by Hadi Partovi*
- *The early days by Steve Wozniak*
- *Hack your life in 48 hours by Dave Fontenot*
- *You Should Learn to Program by Christian Genco*

Programar es un arte y el programador debería ser un artesano, aunque también existen otros puntos de vista menos románticos sobre nuestra profesión.

Las máquinas y los sistemas son geniales haciendo una única cosa, seguir pasos…. La responsabilidad de todo programador en relación a las máquinas es ser capaz de guiarlas con las instrucciones más precisas.

A diferencia de un artesano convencional, el de código, puede evaluar la belleza de un código basándose en ciertos criterios. No se trata solo de que la máquina logre el objetivo deseado, se trata también de no llegar a ese objetivo con un código feo, desastroso e ineficiente.

En programación existen criterios como consistencia, rendimiento, integridad, *complejidad ciclomática*, *nivel de abstracción*, etc… Estos criterios determinan la calidad de un código.

Esto a su vez nos lleva al análisis de nuevos problemas relacionados con la legibilidad y el mantenimiento de nuestro código. El balance es muy complicado y por ello un programa jamás llega a ser perfecto, recuerda que al final, el programador sigue siendo humano…



Programar requiere aprender una nueva forma de pensar

Mucha gente omite este paso y acaba perdida en un infinito desierto de agonía y desesperación cuando se estancan… y se ven incapaces de resolver ciertas situaciones que surgen en el día a día de un programador.

Cosas como la *metaprogramación* y el pensamiento abstracto serán dos objetivos loables a los que debes aspirar como buen artesano del código.

No te preocupes querido lector, porque tus hijos no sufrirán este tormento, ya que están aprendiendo programación de la manera más ágil y divertida, centrándose en el pensamiento abstracto y dejando los lenguajes de programación para más adelante… cosas como *Scratch* o *Bitbloq* se encargan de ayudarles con esta ardua tarea de una manera amena.

Pero para los adultos es otra historia… por eso, antes de aterrizar en el siguiente capítulo, deberíamos emplear un poco más de tiempo aprendiendo las bases de la programación con pseudocódigo.

Las máquinas procesan el mundo de manera binaria, no existen las escalas de grises, por eso el programador debe asumir que tiene que interiorizar y empatizar con la única forma de ver el mundo que tienen las máquinas. Debe aprender a pensar de una manera más "binaria", esto traerá consigo beneficios más allá de la computación, como es aprender a resolver problemas complejos, dividir tareas, etc… y así aumentará enormemente su creatividad. Aprenderá de los errores como nunca antes, tendrá nuevos retos cada día y cuando los supere, la satisfacción será increíble…

Aprender a programar no tiene edad.

Pseudocódigo

El *pseudocódigo* nos permite aislarnos del lenguaje sobre el que finalmente deseamos programar y nos centra en la base lógica más pura de la programación, lo que nos permite adentrarnos en el problema en sí y en su resolución… sin tener que preocuparnos de particularidades de cada lenguaje y su entorno de ejecución.

No vamos a hablar de todas las posibilidades que nos aporta el pseudocódigo, pero si hablaremos de algunas estructuras que resultarán interesantes para nuestro aprendizaje orientado a JavaScript.

Aunque no es necesario realizar nuestro pseudocódigo en un entorno específico, existen programas como *PseInt*, que intentan dotar de un entorno completo a nuestro pseudocódigo.

Por todo ello, los ejemplos que veremos en este capítulo están basados en la sintaxis y funcionamiento de *PseInt*. También incluyo su equiparación con JavaScript a modo de referencia.



¡No te agobies!

Es normal que haya cosas que no entendamos de JavaScript en los próximos ejemplos, para eso tenemos el resto del libro…

Variables

Cuando programamos, podemos crear variables que son pequeños espacios de memoria a los que ponemos un nombre como *"nombre_usuario"*, y que nos permite almacenar datos que posteriormente podremos modificar si deseamos.

- En Pseudocódigo:

```
// Esto es un comentario
nombreUsuario = "Pepe"
nombreOtroUsuario = "Roberta"
```

- Equiparación en JavaScript:

```
// Esto es un comentario
var nombreUsuario = "Pepe";
var nombreOtroUsuario = "Roberta";
```

Interacción con el usuario

Podemos pedir al usuario que nos facilite algún tipo de dato, almacenarlo en una variable y también podemos mostrarle información.

- En Pseudocódigo:

```
// Le mostraremos el siguiente mensaje
Escribir "hola usuario! ¿Cómo te llamas?"

// Almacenamos su respuesta en una variable
Leer nombreUsuario
```

- Equiparación en JavaScript:

```
// Le mostraremos el siguiente mensaje
var mensaje = "hola usuario! ¿Cómo te llamas?";

// Almacenamos su respuesta en una variable
var nombreUsuario = prompt(mensaje);
```

Como podemos ver… a la hora de equiparar este mensaje en JavaScript, he metido ciertos cambios para agilizar el *algoritmo* y adaptarlo al contexto de JavaScript en el navegador.

Estructuras condicionales

Cuando necesitamos que el sistema tome decisiones, debemos plantearle las múltiples opciones en una estructura condicional concreta.

Para que la lógica del sistema pueda decidir, necesitamos reducir todo a comparaciones susceptibles de devolver un *valor booleano (verdadero o falso)*.

Por ejemplo:

- Operaciones matemáticas
  - 1+1 = 2
  - 5 es mayor que 3
  - -5 es menor que cero
- Operaciones lógicas
  - La variable "nombreUsuario" es igual a "Pepe"
  - La variable "nombreUsuario" no está vacía

A la hora de representar las opciones. *PseInt* nos da dos opciones principales. La estructura *Si.. entonces* y *Según*, la diferencia en este contexto está en función del número de opciones.

En el caso de *Si…*
: Entonces solo manejamos dos opciones cuando la condición se cumple, y cuando no se cumple.

En el caso de *Según*
: Podemos manejar multiples valores y con ello multiples opciones sobre las que el sistema deberá decidir.

: Si una condición puede cumplirse en múltiples casos, el sistema se decidirá por la primera e ignorará el resto.

Si… Entonces

- En Pseudocódigo:

```
Si 1 + 1 == 2 Entonces
    Escribir "La suma es correcta"
Sino
    Escribir "La suma es incorrecta"
Fin Si
```

- Equiparación en JavaScript:

```
if (1 + 1 === 2) {
    console.log("La suma es correcta")
} else {
    console.log("La suma es incorrecta")
};
```

Según

- En Pseudocódigo:

```
Escribir "hola usuario! ¿Cómo te llamas?"
Leer nombreUsuario
Segun nombreUsuario Hacer
    "Pepe":
        Escribir "Hola Pepe! Yo te conozco"
    "Lucia":
        Escribir "Hola Lucia! Yo te conozco"
    De Otro Modo:
        Escribir "Eres nuevo... Bienvenido!"
    Fin Segun
```

- Equiparación en JavaScript:

```
var nombreUsuario = prompt("hola usuario! ¿Cómo te llamas?")

switch(nombreUsuario) {
    case "Pepe":
        console.log("Hola Pepe! Yo te conozco");
        break;
    case "Lucia":
        console.log("Hola Lucia! Yo te conozco");
        break;
    default:
        console.log("Eres nuevo... Bienvenido!");
}
```

Bucles

En ocasiones nuestro algoritmo requerirá ejecutar una serie de instrucciones de manera repetitiva hasta que cierta circunstancia cambie. Esto en programación se conoce como bucles, básicamente tenemos tres tipos.

Al existir tres tipos, surge la duda de saber si realmente existen diferencias significativas entre ellos.

Como es normal existen ciertas diferencias en el manejo de las interacciones, pero todos los bucles mantienen una estructura y funcionamiento común que se divide en dos partes:

Condición
: Nos permite definir cual es la condición que debe cumplirse para que nuestro programa decida si hay que *entrar/seguir* en bucle o *ignorarlo/salir*.

  El dominio de esta parte del bucle nos asegurará no caer en bucles infinitos o bucles que nunca serán ejecutados.

Instrucciones
: Es un conjunto de comandos que serán ejecutados una vez el bucle haya validado las condiciones.

Tipos de bucles

- **While (Mientras)**

  Está pensado para aquellos bucles que requieran de una condición lógica y no matemática para validar la iteracción.



Aunque lógicamente podremos mezclar entre las diversas condiciones (matemáticas y lógicas).

  - En Pseudocódigo:

```
vuelta <- 0
Mientras vuelta < 100 Hacer
    Escribir vuelta
    vuelta = vuelta + 1
Fin Mientras
```

  - Equiparación en JavaScript:

```
var vuelta = 0;

while(vuelta < 100){
    console.log(vuelta)
    vuelta++ // vuelta = vuelta +1
}
```

- **For (Para)**

  Está diseñado específicamente para trabajar con series de números (longitud de cadena, elementos de un array, propiedades de un objeto, etc…) y nos provee de una estructura más sencilla para gestionar el flujo.

  - En Pseudocódigo:

```
Para v_numerica<-v_inicial Hasta vuelta < 100 Con Paso vuelta = vuelta + 1 Hacer
    Escribir vuelta
Fin Para
```

- Equiparación en JavaScript:

```
for(var vuelta = 0; vuelta < 100; vuelta++ ){
    console.log(vuelta);
}
```

- **Do… While (Repetir… Hasta Que)**

  Este bucle funciona de una manera diferente. Altera el orden al que estamos acostumbrados, ya que primero ejecutará el código una vez y luego evaluará si debe repetirse.

  - En Pseudocódigo:

```
dato <- Falso
Repetir
    Escribir "Al menos una vez..."
Hasta Que Dato == Verdadero
```

  - Equiparación en JavaScript:

```
var Dato = false;

do {
    console.log("Al menos una vez...");
} while(Dato)
```

Funciones o subprocesos

Las funciones son estructuras de código que nos permiten reutilizar en gran medida nuestro código, gracias a que encapsulan instrucciones en su interior.

Las funciones pueden realizar dos acciones principales, de manera independiente o interrelacionadas.

Pueden modificar el valor de las variables que están fuera de la función (ámbito global) o retornando un nuevo valor al final de su ejecución.

Otra de las características que hacen de las funciones estructuras muy valiosas, es el uso de argumentos. Esto es una especie de variable que se declara junto a la ejecución y que no tiene ningún valor hasta que no llamemos a la función.

A la hora de llamar a la función podemos pasarle parámetros que son valores con los que la función podrá realizar todo tipo de operaciones.

Todo sistema, incluso *PSeInt*, dispone de funciones que han sido creadas para facilitar nuestro trabajo, como son las funciones matemáticas que nos permiten calcular, desde valores absolutos hasta logaritmos, pasando por funciones de azar para generar números aleatorios.

También existen funciones propias según el tipo de dato con el que trabajemos, como es el caso de las cadenas de texto, donde podemos convertir su valor a números o saber la longitud total de los caracteres que lo componen, etc…

Hablaremos de todo esto en próximos capítulos.

Operadores especiales

En todo lenguaje de programación encontraremos ciertos operadores comunes que nos facilitarán tareas, como son:

Algebraicos
: Relacionados con operaciones matemáticas (sumar, restar, multiplicar, dividir…).

Relacionales
: Relacionados con comparaciones entre elementos (distinto a, igual que, menor que, mayor o igual que…).

Lógicos
: Permiten encadenar validaciones entre elementos. Y (conjunción) y O (disyunción).

  Por ejemplo: ((es usuario) Y (está registrado)) O (es un invitado).

Capítulo 3 - console.log("Hola Mundo");

JSHint

Es una herramienta clave para nuestro día a día, ya que nos permite detectar errores comunes en el código JavaScript que desarrollemos.

Para usuarios más avanzados es recomendable utilizar *esLint*, ya que nos proporciona múltiples configuraciones avanzadas.

JSHint puede ser utilizado directamente desde la web o como un plugin más en tu editor favorito.



Más información:

- *JSHint Online*
- *JSHint About*
- *JSHint Repository*

Consola

La herramienta más utilizada será la consola de JavaScript, que es una de las múltiples utilidades que nos ofrecen los navegadores para depurar nuestro código.

Si ya desarrollas habitualmente utilizando CSS y HTML, estarás familiarizado con este conjunto de herramientas y su potencial, si no deberías *seguir estos sencillo pasos*.

Yo personalmente, prefiero utilizar Google Chrome aunque sin duda Mozilla Firefox, también es una buena opción para depurar y desarrollar nuestro código.

Para hacer uso de la consola… necesitamos llamar al objeto *console*.



Más información:

- *@ChromeDevTools en Twitter*
- *Chrome DevTools*

Métodos destacados:

Hacemos una recopilación simplificada de los métodos más utilizados, aunque existen muchos más.

- ***.assert()***

  Aparece un mensaje de error en la consola si la afirmación es falsa.

```
var controlador = false;
console.assert(controlador, "\"controlador\" es igual a \"false\"");
```

- ***.clear()***

  Limpia la consola.

```
console.clear()
```

- ***.dir()***

  Retorna una lista interactiva de las propiedades de un objeto.

```
console.dir(document.body);
```

- ***.dirxml()***

  Retorna una representación HTML del objeto.

```
console.dirxml(document.body);
```

Agrupadores

- ***.group()***

  Crea un grupo de mensajes de consola.
- ***.groupCollapsed()***

  Crea un grupo de mensajes de consola minimizados por defecto.

```
console.groupCollapsed("bucleFor");
for(var i=100; i>0; i--){
    console.info("Iteración numero %i", i)
}
console.groupEnd();
```

- ***.groupEnd()***

  Cierra el grupo de mensajes.

Tablas

- ***.table()***

  Muestra los datos dentro de una tabla.

```
var lenguajes = [
    { nombre: "JavaScript", extension: ".js" },
    { nombre: "TypeScript", extension: ".ts" },
    { nombre: "CoffeeScript", extension: ".coffee" }
];

console.table(lenguajes);
console.table(lenguajes, "extension");
```

Gestión del tiempo

- ***.time()***

  Inicia un contador en ms.
- ***.timeEnd()***

  Para el contador y devuelve el resultado.

```
console.time("Medición de miArray");
var miArray = new Array(1000000);
for (var i = miArray.length - 1; i >= 0; i--) {
 miArray[i] = new Object();
};
console.timeEnd("Medición de miArray");
```

Notificadores

- ***.log()***

  Saca un mensaje por consola.

```
console.log("Hola en formato clásico");
```

- ***.info()***

  Saca un mensaje por consola con un estilo informativo.

```
console.info("Hola en formato informativo");
```

- ***.warn()***

  Saca un mensaje por consola con un estilo alerta.

```
console.warn("Hola en formato alerta");
```

- ***.error()***

  Saca un mensaje por consola de error, con los mismos estilos, creando confusión. No se recomienda su uso.

```
console.error("Hola en formato error");
```

Formateadores

Formato	Descripción
%s	Cadena
%d o %i	Número entero
%f	Decimal
%o	DOM
%O	Objeto js
%c	CSS

Ejemplos:

- *%o* para estructuras del DOM:

```
var parrafos = document.getElementsByTagName("p");
console.log("DOM: %o", parrafos);
```

- *%O* para objetos JS:

```
var objeto = {"nombre":"Yo","Apellido":"Mismo"};
console.log("Objeto: %O", objeto);
```

- Usando CSS:

```
console.log('Esto es aburrido...');
console.log('%c Pero se puede mejorar fácilmente! ', 'background: #3EBDFF; colo\
r: #FFF; font-size:25px;');
```

Caracteres especiales:

Incluyendo ciertos caracteres especiales en nuestras cadenas de texto, podemos maquetar el resultado que se imprimirá por la consola.

- `\t` *Tabulador*
- `\'` *Comillas Simples*
- `\"` *Comillas Dobles*
- `\n` *Salto de línea*

```
console.log("Hasta aquí... todo correcto. Ahora vamos a \"tabular\":\tves? Ya est\
amos más lejos.\n\'Otra linea ;-)\'")
```

Comentarios



Los comentarios nos ayudan a entender nuestro código, en ocasiones podemos utilizarlos para silenciar temporalmente partes del código.

Algunos desarrolladores aconsejan utilizar *JSDoc*, como un sistema para determinar como debe comentarse de manera sistemática nuestro código.

Al utilizar *JSDoc* puedes exportar esos comentarios como un fichero Html, que nos permitirá documentar nuestro proyecto fácilmente.

Como regla general es mejor poner comentarios, que no ponerlos. También puedes usar los comentarios de una forma más eficiente si referencias (urls) a la documentación en el caso de las librerías.

- **Una línea**

```
// Comentario en una línea
```

- **Multiples líneas**

```
/*
Una Línea....
Otra linea...
Etc...
*/
```

- **JSDoc**

```
/**
 * Represents a book.
 * @constructor
 * @param {string} title - The title of the book.
 * @param {string} author - The author of the book.
 */
function Book(title, author) {
}
```

Nombres de variables

Las variables tienen unas reglas muy especificas que debemos seguir a la hora de designar sus nombres.

- Debemos usar nombres que expliquen que aporta esa variable a nuestro código.
- No se pueden usar espacios.

```
var con espacios = 1;
```

- No usar un número delante.

```
var 1numero = 1;
```

- Evitar símbolos y *snake_case*.

```
var con_guiones_bajos = 1; //snake_case es poco común en JavaScript
var dame$ = 1; // en serio?
```

- Usar *camelCase*.

```
var otraOpcion = 1;
var opcionCon123123 = 1;
```



Truco

Si estas realizando cambios en un código que sigue una pauta distinta a cameCase. ¡Síguelo!

La coherencia en el código es más importante, recuérdalo.

Tipos de variables

Un primer paso para dominar los tipos de variables es utilizar el operador *typeof*, y conocer su *especificación*.



Tipado Debil vs. Tipado Fuerte

Las variables en JavaScript no necesitan ser declaradas teniendo en cuenta *el tipo de dato* que contienen, como sucede en otros lenguajes

Es importante conocer los tipos de *variables disponibles*:

- ***Undefined***

```
typeof undefined
typeof noDefinido
typeof tampocoCreado
```

- ***Object***

```
typeof null
typeof [15, 4]
typeof new Date()
typeof {a:1}
```

- ***Boolean***

```
typeof false
typeof true
typeof Boolean(false)
```

- ***Number***

```
typeof 3
typeof 3.14
typeof NaN
typeof Infinity
```

- ***String***

```
typeof "hola"
```

- ***Function***

```
typeof function(){}
```

Existe un tipo más… *Symbol* pero pertenece a *ECMA6*, es por ello, que no lo desarrollamos aquí.

Matemáticas Básicas

Las matemáticas básicas en JavaScript son muy similares a cualquier lenguaje.

El único operador matemático que puede resultarnos extraño es *el módulo*.

```
var suma = 5 + 4;
var resta = 10 - 6;
var multiplicacion = 3 * 3;
var division = 6 / 2;
var modulo = 43 % 10;
```



Trabajando con Decimales

En JavaScript el decimal 1.0 y el entero 1, son iguales.

```
console.log((5.0+1) === (5+1)) // true
```

Matemáticas Básicas (Agrupando operaciones)

Con los paréntesis fácilmente se pueden priorizar operaciones.

El orden de ejecución empieza por los parentesis, luego multiplicaciones y divisiones… para terminar con sumas y restas.

```
var expresion1 = (3 + 7) * 10;
var expresion2 = (-56 * 6) - 74 * -25;
var expresion3 = (3 * 3) + 10 - 12 / 2;
var expresion4 = 44 + (83 % (33 + 100));
var expresion5 = -145 + (500 / 10 - 5) + 10 * 10 ;
```

Matemáticas Básicas (crecimiento y decrecimiento)

El crecimiento y decrecimiento, puede suceder antes o después que el valor de la variable sea leído por el navegador, en función de dónde se sitúa el operador (++antes o después++).

```
var numero = 5;

console.log(--numero); // 4
console.log(numero--); // 5 (luego 4)
console.log(++numero); // 6
console.log(numero++); // 5 (luego 6)
```

Operadores de asignación

Estos operadores, nos permiten alterar de manera sencilla y controlada el valor de una variable.

- = **Asignación**

```
var x = 1;
var y = 2;
x = y;
console.info("\"x\" vale ", x);
```

- += **Suma**

```
var x = 1;
var y = 2;
x += y; // x = x + y
console.info("\"x\" vale ", x);
```

- -= **Resta**

```
var x = 1;
var y = 2;
x -= y; // x = x - y
console.info("\"x\" vale ", x);
```

- *= **Multiplicación**

```
var x = 1;
var y = 2;
x *= y; // x = x * y
console.info("\"x\" vale ", x);
```

- /= **División**

```
var x = 1;
var y = 2;
x /= y; // x = x / y
console.info("\"x\" vale ", x);
```

- %= **Módulo (Resto)**

```
var x = 1;
var y = 2;
x %= y;  // x = x % y
console.info("\"x\" vale ", x);
```

Jugando con variables

Si ejecutas el siguiente fragmento de código, verás lo fácil que resulta perder el control de nuestras variables, si no tenemos mucho cuidado.

```
var empezoComo3 = 3;
era3();

empezoComo3 = 35;
era3();

empezoComo3 = empezoComo3 + 30;
era3();

empezoComo3 += 4;
era3();

empezoComo3++;
era3();

empezoComo3 -= 12;
```

```
era3();

empezoComo3--;
era3();

empezoComo3 *= 10;
era3();

empezoComo3 /= 11;
era3();

empezoComo3 += "texto";
era3();

empezoComo3 += 20;
era3();

empezoComo3++;
era3();


function era3 () {
    console.log("empezoComo3 debería ser 3, ahora su valor es " + empezoComo3);
};
```

Interacción Básica con el Usuario

Puedes interactuar con el usuario desde JavaScript, utilizando unos métodos sencillos que ya vienen implementados en todos los navegadores. Esto hace unos años era muy común y poco a poco se convirtió en una mala práctica.



El propio navegador, ofrece al usuario la posibilidad de bloquear esta interacción. Lo ideal es gestionar esta interacción con el usuario desde el HTML, como veremos en próximos capítulos.

En el caso de depuración y prototipado de código puede ser un recurso muy útil.

- ***Window.alert()***

  Muestra nuestro mensaje en una ventana.

```
alert("¡Bienvenido a esta web!");
```

- ***Window.confirm()***

  Pregunta al usuario y ofrece dos botones que se traducen en valores booleanos. Aceptar (true) y cancelar o cerrar la ventana (false).

```
confirm("¿Está seguro que desea abandonar esta web?");
```

  Ejemplo:

```
var respuesta = confirm("presiona un botón!");
if (respuesta === true) {
    console.log("Has aceptado!");
} else {
    console.log("Has cancelado");
}
```

- ***Window.prompt()***

  Pregunta al usuario y permite la escritura devolviendo el mensaje.

```
prompt("¿Como te llamas?");
```

  Registremos los datos en una variable:

```
var nombre = prompt("¿Como te llamas?");
```

Parte II - Mecánica del lenguaje

Capítulo 4 - Comparadores

Operadores de Comparación

Muchos de los comparadores, mayor que, menor que, igual que, etc… ya los conocemos. Pero en JavaScript tenemos un comparador adiccional, que nos permite comparar entre dos elementos, no solo por su valor, también por su tipología.

```
var mayorQue = 100 > 10;
var menorQue = 10 < 100;
var mayorIgual = 100 >= 10;
var menorIgual = 10 <= 100;
var igual = 10 == 10;
var igualTotalmente = 10 === 10; // Ojo! Usamos también ===
var noIgual = 100 != 10;
```

Como regla general, será imperativo utilizarlo siempre que comparemos entre valores el ===, ya que debemos asegurarnos que no nos den "gato por liebre", confirmando que son iguales ambos datos en valor y tipo de dato.

Veamos el siguiente ejemplo:

```
var igual = 10 == 10; // true
var igualCadena = 10 == "10"; //true
var igualTotalmente = 10 === 10; // true
var igualTotalmenteCadena = 10 === "10"; //false

var noIgual = 100 != 10; // true
var noIgualCadena = 100 != "100"; // false
var noIgualTotalmente = 100 !== 10; // true
var noIgualTotalmenteCadena = 100 !== "100"; // true
```

Operadores Lógicos

Podemos concatenar validaciones y así evaluar expresiones más complejas.

En JavaScript disponemos de *&& (todo debe resolverse como verdadero)* y || *(Al menos uno de los elementos debe ser verdadero)*

- **AND(&&)**

```
console.log(true && true);      // true
console.log(true && false);     // false
console.log(false && false);    // false
console.log(false && true);     // false
```

- **OR(||)**

```
console.log(true || true);      // true
console.log(true || false);     // true
console.log(false || false);    // false
console.log(false || true);     // true
```

Podemos también utilizar lo aprendido hasta ahora con operaciones matemáticas:

```
var ex1 = true && true;          // true
var ex2 = (2 == 2) && (3 >= 6); // false
var ex3 = (2>3) || (17 <= 40);  // true
var ex4 = false || false;        // false
```

Todo puede ser booleano

En JavaScript todo puede ser comparado en términos booleanos, especialmente si hacemos uso de la función *Boolean()*.

En general podemos saber cómo responderá JavaScript, si tenemos en cuenta el *"valor real"* de la expresión que deseamos validar.

Por ejemplo, una cadena de texto vacía *("")* no aportará valor real, por lo que será *false*.

Importante remarcar que el *0* y *-0* son false

- **Valor "real" es *true*:**

```
console.log("valor boleano de \"JS!\":", Boolean("JS!\""));
console.log("valor boleano de 1235:", Boolean(1235));
console.log("valor boleano de -1235:", Boolean(-1235));
console.log("valor boleano de un objeto:", Boolean({saludo: "hola"}));
console.log("valor boleano de un array:", Boolean(["platano", -1, false]));
console.log("valor boleano de un array:", Boolean(function(){}));
```

- **Sin valor "real" es *false*:**

```
console.log("valor boleano de \"\":", Boolean(""));
console.log("valor boleano de 0:", Boolean(0));
console.log("valor boleano de -0:", Boolean(-0));
console.log("valor boleano de null:", Boolean(null));
console.log("valor boleano de undefined:", Boolean(undefined));
console.log("valor boleano de NaN:", Boolean(NaN));
```

Asignación por igualdad

Una manera ágil y dinámica de asignar valor a las variables, es haciendo uso de la asignación por igual. Básicamente almacenamos el resultado booleano de una comparación en una variable que podremos utilizar más adelante.

```
var administrador = 'Yo mismo';
var esAdministrador = (administrador === 'Yo mismo');
console.log(esAdministrador); // true
```



If

Como ya vimos en pseudocódigo, una de las estructuras más útiles a la hora de gestionar el flujo de nuestra aplicación es el uso de condicionales.

If nos permite evaluar una condición y actuar en consecuencia.

Las instrucciones que se encuentran entre corchetes, se ejecutan solo cuando la condición se cumple.



Gestiona la complejidad

Dentro de una estructura condicional podemos meter más estructuras condicionales, además de otras estructuras típicas del lenguaje… esto se conoce como *anidación*.

Cuando la anidación es desmesurada y poco óptima suele elevarse mucho la complejidad ciclomática de nuestro código. De esto hablaremos en próximos capítulos.

```
if(1 === 1){
        console.log("1 es igual a 1 y por eso me ejecuto")
}

if(1 === "1"){
        console.log("No son del mismo tipo y por eso... este texto jamás será mostrado e\
n la consola.")
}
```

If… else

En ciertas ocasiones, necesitaremos que nuestro script elija entre dos caminos en función de si una premisa es *verdadera* o *falsa*, por eso JavaScript nos permite utilizar *else*.

El programa solo puede ejecutar una de las dos opciones, por tanto una parte del código quedará sin ejecutarse.

```
console.log("pase lo que pase... esto se ejecutará")
if (true) {
    console.log("true, por eso me ejecuto");
} else {
    console.log("false, por eso me ejecuto");
}
console.log("pase lo que pase... esto se ejecutará también")
```

Else if…

Cuando necesitemos verificar diversas opciones lo más fácil es utilizar *else if*, así evitamos tener que anidar en exceso.

Es importante recordar que la primera condición válida será la que use el interprete (navegador), y que ignorará todas las demás.

```
var condicion = 2;
if (condicion == 1) {
    console.log("1, por eso me ejecuto");
} else if (condicion == 2){
    console.log("2, por eso me ejecuto");
} else {
    console.log("no es 1 o 2, por eso me ejecuto");
}
```

Switch

JavaScript dispone de una opción más para crear estructuras condicionales. *Switch* permite crear estructuras más optimizadas para cubrir un amplio abanico de casos posibles.

Por otra parte *Switch* tiene varias desventajas notables:

- No tiene una sintaxis sencilla
- Es importante utilizar y comprender conceptos propios como *case*, *break* o *default*.
- En ocasiones puede ser difícil de depurar.
- Erróneamente se piensa que *Switch* es exageradamente más rápido que *if-else*, aunque una *simple prueba demuestra lo contrario*.
- **Entendiendo la estructura:**

```
switch(expresión) {
    case n1:
        //Código
        break;
    case n2:
        //Código
        break;
    default:
        //Código
}
```

- **Trabajando con "casos únicos":**

```
var nombre = "";
switch (nombre) {
  case "Pepe":
    console.log("Hola Pepe");
    break;
  case "Luis":
    console.log("Hola Luis");
    break;
  case "Antonio":
    console.log("Hola Antonio");
    break;
  default:
    console.log('Ninguno de los nombres que pensamos... es '+nombre);
}
```

- **Trabajando con "múltiples coincidencias":**

```
var nombre = "";
switch (nombre)
{
   case "Pepe":
   case "Luis":
   case "Antonio":
       alert('Hola '+nombre);
       break;

   default:
       console.log('Ninguno de los nombres que pensamos... es '+nombre);
}
```

Operador Ternario

El *operador ternario (?:)* nos ofrece una manera propia de hacer estructuras condicionales.

Este operador simplifica mucho la sintaxis propia de los condicionales, pero se desaconseja su uso para operaciones o evaluaciones múltiples.



Si estás empezando con JavaScript, debes tener en cuenta, que es un operador que resulta difícil de recordar al principio. Siendo bastante común su uso especialmente en proyectos desarrollados con Node.js.

Estructura

- **Entendiendo la estructura:**

```
condicion ? expresion1 : expresion2
```

- **Múltiples expresiones (desaconsejado):**

```
condicion ? (
    expresion1,
    expresion2,
    otraexpresion
) : (
    expresion1,
    expresion2,
    otraexpresion
);
```

- **Evaluaciones múltiples (muy desaconsejado):**

```
condicion ? expresion1 : condicion2 ? expresion1 : expresion2;
```

Ejemplos

- **Sencillo:**

```
var esMiembro = true;
console.info("El pago son  " + (esMiembro ? "20.00€" : "50.00€"));
```

- **Evaluación múltiple (muy desaconsejado):**

```
var esMiembro = true;
var esAdulto = true;
console.info(esMiembro ? "El pago son 20.00€" : esAdulto ? "Puedes enviar la so\
licitud cuando quieras" : "Tienes que esperar aún. Lo siento.");
```

- **Múltiples expresiones (desaconsejado):**

```
var mensaje,
esMiembro = true;

esMiembro ? (
    mensaje = "El pago son 20.00€",
    console.info(mensaje)
) : (
    mensaje = "El pago son 50.00€",
    console.info(mensaje)
);
```

Capítulo 5 - Bucles

En JavaScript existen diversos bucles que podremos utilizar en función de las necesidades de nuestro programa.

Ahora veremos aquellos bucles que son de uso genérico, pero existen variaciones del bucle *for* específicas para Arrays *forEach*) y Objetos *For… in*) que veremos en próximos capítulos.

Para el caso que os preocupe el rendimiento de vuestros bucles os dejo *este link* con unos *benchmarks* bastante interesantes.

While

El primer bucle que veremos está específicamente diseñado para funcionar de manera constante, mientras una condición dada siga cumpliéndose.

Este bucle está pensado para simplificarnos la sintaxis cuando el control del bucle no se realiza mediante operaciones matemáticas (mayor que… menor que…).

Funcionamiento

- **Estructura:**

```
while (-Condición-) {
    -Instrucciones-
};
```

- **Ejemplo:**

```
var condicion = true
while (condicion) {
    console.log("hola");
    condicion = false;
};
```

For

Este bucle presenta una estructura optimizada para controlar la ejecución de la iteración de manera numérica.

El bucle *For* se divide en tres partes separadas por un punto y coma.

Expresión inicial
Será todo aquello que se ejecutará al iniciarse el bucle.
Condición
Será evaluada antes de cada iteración. Este es el único parámetro obligatorio.
Expresión de actualización
Se ejecutará al final de cada iteración.

Funcionamiento

- Estructura:

```
for (-Expresión inicial-; -Condición-; -Expresión Actualización-) {
    -Instrucciones-
};
```

- Ejemplo clásico:

```
for (var i = 0; i < 10; i++) {
    console.log(i);
}
```

Do… While

El tercer bucle que veremos en este capítulo es *Do… While* que se diferencia de todos los demás en que se ejecuta primero y se evalúa después.

Al utilizar este bucle nos aseguramos que aunque la condición no se cumpla… el código se ejecuta al menos una vez.

Funcionamiento

- Estructura:

```
do{
   -Instrucciones-
} while (-Condición-);
```

- Ejemplo:

```
var i = 0;
do {
   i++;
   console.log(i);
} while (i < 10);
```



Importante:

Al menos se ejecutará una vez, aunque la premisa no sea verdadera.

```
do{
   console.warn("me ejecuto")
} while (false);
```

Break y *Continue*



Dominando la eficiencia

Dos sentencias clave en JavaScript, son *break* y *continue*, que nos permiten romper o alterar el flujo normal de nuestra aplicación.

Pueden utilizarse también con las estructuras condicionales que vimos en el capítulo anterior.

- **Continue**

  Nos permite saltar parte del bucle.

```
for (var i = 0; i < 10; i++) {

    // Salta el 5 y sigue...
    if (i === 5) {
            continue;
    }

    console.log("El valor de i es "+i);
}
```

- **Break**

  Nos permite salir del bucle.

```
for (var i = 0; i < 10; i++) {

    // Llega a 5 y sale.
    if (i === 5) {
            break;
    }

    console.log("El valor de i es "+i);
}
```

Errores comunes

Bucle infinito

Es un error muy común y deberías evitarlo a toda costa. Suele ocurrir cuando no tenemos una estructura de control funcionando adecuadamente.

```
while (true) {
    console.log("Este texto se imprime hasta el infinito...");
};
```

Bucle que no se ejecutará

Igualmente es un error más sutil, pero por definición **un código que jamas se ejecuta… ¡sobra!**

```
while (false) {
    console.log("Este texto jamas se imprimirá...");
};
```

Ejemplo:

```
var control = 1;
while (control <= 10) {
    console.log(control);
    control++;
};
```

Usos Avanzados

Decrecimiento:

Es bastante común utilizar este tipo de bucles, cuando estamos trabajando sobre la representación dinámica de elementos en el DOM.

```
for (var i = 10; i > 0; i--) {
    console.log(i);
}
```

Anidación:

Aunque resulta muy tentador… debemos de evitar en la medida de lo posible la anidación en JavaScript. Por regla general anidar hasta dos o tres niveles es aceptable.

```
for (var i = 0; i < 10; i++) {
    console.log("Estoy en el primer bucle.");
    for (var j = 10; j > 0; j--) {
        console.log("Estoy en el segundo bucle")
        console.log("Vuelta: " + i + " - " + j);
    }

}
```

Exprimiendo el For

Si dominamos las tres partes básicas que componen el bucle *for*, podemos plantearnos estructuras tan complejas como ésta.

```
for (var i = 0, x = 1, z = 2, tope = 10; i <= tope; x *= z, i++ ) {
    console.log("i vale "+i+", x vale "+x+", z vale "+z);
}
```



Por lo general la *variable i* suele reservarse para controlar las iteraciones del bucle, aunque podamos utilizar otros nombres… esto suele estar muy aceptado por la comunidad y se considera una buena práctica.

No refactorizar

En ocasiones, el código evoluciona y no repasamos lo que habíamos construido previamente… llegando a situaciones tan monstruosas como ésta.

- Antes de refactorizar:

```
// Código
for (;i === true;) {
    // Más Código
}
```

- Después de refactorizar:

```
// Código
while (i) {
    // Más Código
}
```

Capítulo 6 - Números y fechas

JavaScript al igual que otros muchos lenguajes, permite el trabajo matemático, aún no siendo un lenguaje especialmente diseñado para científicos y matemáticos.

A lo largo de este capítulo veremos que en JavaScript existen dos formas básicas de trabajar con los *"números"*, por un lado haciendo uso de *Number*, y por otro con *Math*, en función de la complejidad de nuestros algoritmos.

Por lo general, es interesante utilizar ciertas librerías clave que nos facilitaran mucho el desarrollo cuando trabajamos con temas numéricos, os las presentaré a lo largo de este capítulo.

Numbers

A la hora de trabajar con números, puede darse el caso de encontrarnos con la *notación científica* en vez del formato al que estamos acostumbrados. Su valor y tipo de variable se mantienen aunque se represente de forma diferente.

Si deseamos trabajar con unidades monetarias, podemos hacer uso de la librería *accounting.js*.

Propiedades

- ***.MAX_VALUE***

  Esta propiedad nos retornará el número (positivo) más grande representable en JavaScript:

```
var numero1 = 1.7976931348623157e+308;
var numero2 = 1.7976931348623157e+310;

function verificarValorMaximo(num){

            if (num <= Number.MAX_VALUE) {
              console.log("El número no es infinito");
            } else {
              console.log("El número es infinito");
    }

}

verificarValorMaximo(numero1);
verificarValorMaximo(numero2);
```

- ***.MIN_VALUE***

  Nos retornará el número (negativo) más pequeño representable (5e-324)

```
var numero1 = 5e-323;
var numero2 = 5e-326;

function verificarValorMinimo(num){
    if (num >= Number.MIN_VALUE) {
        console.log("El número no es infinito");
    } else {
        console.log("El número es infinito");
    }
}

verificarValorMinimo(numero1);
verificarValorMinimo(numero2);
```

- ***.POSITIVE_INFINITY***

  Retornará el valor de la propiedad del objeto *global Infinity*, es decir *Infinity*.

```
var numeroMaximo = Number.MAX_VALUE * 2
console.log(numeroMaximo);

if (numeroMaximo === Number.POSITIVE_INFINITY) {
 numeroMaximo = 0;
}
console.log(numeroMaximo);
```

- ***.NEGATIVE_INFINITY***

  Retornará el valor negativo de la propiedad del objeto *global Infinity*, es decir *-Infinity*.

```
    var numeroMinimo = (-Number.MAX_VALUE) * 2
    console.log(numeroMinimo);

    if (numeroMinimo === Number.NEGATIVE_INFINITY) {
     numeroMinimo = 0;
    }
    console.log(numeroMinimo);
```

- ***.NaN***

  En ocasiones al realizar operaciones matemáticas de forma errónea podemos encontrarnos con que el valor de nuestra variable ha sido sustituido por *NAN (Not A Number)*. Este es un proceso irreversible que nos obligará a reasignar el valor de la variable posteriormente. Hasta la llegada de ECMA6 la gestión y detección de estos valores era compleja, pero con el nuevo método *isNaN()* podemos solventarlo.

```
      NaN === NaN;         // false
      Number.NaN === NaN; // false
      isNaN(NaN);          // true
      isNaN(Number.NaN);   // true
```

Métodos

Hacemos una recopilación simplificada de los métodos más utilizados, aunque existen muchos más.

- ***.toExponential()***

  Retorna una cadena de texto con el valor de los números en formato de potencia.

```
var numObj = 77.1234;

console.log(numObj.toExponential());  // 7.71234e+1
console.log(numObj.toExponential(4)); // 7.7123e+1
console.log(numObj.toExponential(2)); // 7.71e+1
console.log(77.1234.toExponential()); // 7.71234e+1
```

- ***.toFixed()***

  Retorna una cadena de texto con los números decimales.

  Aunque no es la mejor forma, también podemos redondear el valor.

```
var numObj = 12345.6789;
numObj.toFixed();       //'12346' redondeo
```

Podemos alterar la longitud (número de decimales) con el argumento que pasamos al método.

```
var numObj = 12345.6789;
numObj.toFixed();       //'12346' redondeo
numObj.toFixed(1);      //'12345.7'
numObj.toFixed(6);      //'12345.678900' Se añaden ceros en caso necesario
(1.23e+20).toFixed(2);  //'123000000000000000000.00'
(0).toFixed(2);         //'0.00'
2.34.toFixed(1);        //'2.3' redondeo
```

Al utilizar valores decimales negativos es importante albergarlos dentro de los paréntesis para que respete el tipo de dato también.

```
-2.34.toFixed(1);       //-2.3 Números negativos no son devueltos como cadenas.
(-2.34).toFixed(1);     //'-2.3' Con paréntesis se salta la limitación
```

- ***.toLocaleString()***

  Retorna una cadena con el valor representado en lenguaje local.

```
var numero = 3500;
// En Local
console.log(numero.toLocaleString());
// En Árabe
console.log(numero.toLocaleString('ar-EG'));
// En Chino decimal
console.log(numero.toLocaleString('zh-Hans-CN-u-nu-hanidec'));
```

- ***.toPrecision()***

  Devuelve un numero con los decimales deseados pero sin redondear.

```
var numero = 5.123456;
console.log(numero.toPrecision());    // 5.123456
console.log(numero.toPrecision(5));   // 5.1235
console.log(numero.toPrecision(2));   // 5.1
console.log(numero.toPrecision(1));   // 5
console.log((1234.5).toPrecision(2)); // 1.2e+3 (En ocasiones )
```

- ***.toString()***

  Devuelve una cadena con el número en la base (hexadecimal, binaria…) que determinemos.

```
console.log((17).toString());     // '17'
console.log((17.2).toString());   // '17.2'
console.log((-0xff).toString(2)); // '-11111111'
console.log((254).toString(16));  // 'fe'
```

- ***.parseFloat()***

  Devuelve un número decimal o entero partiendo de una cadena.

```
Number.parseFloat("3.14");          // 3.14
Number.parseFloat("314e-2");        // 3.14
Number.parseFloat("0.0314E+2");     // 3.14
Number.parseFloat("3.14textos..."); // 3.14
Number.parseFloat("1textos...");    // 1
```

- ***.parseInt()***

  Devuelve un número entero partiendo de una cadena.

  Además, nos permite seleccionar la base númerica sobre la que trabajaremos (binaria con 2, hexadecimal con 16, etc…). Por defecto se utiliza la base 10 (decimal).

```
parseInt(" 0xF", 16);    // 15
parseInt(" F", 16);      // 15
parseInt("17", 8);       // 15
parseInt(021, 8);        // 15
parseInt("015", 10);     // 15
parseInt(15.99, 10);     // 15
parseInt("15,123", 10);  // 15
parseInt("FXX123", 16);  // 15
parseInt("1111", 2);     // 15
parseInt("15*3", 10);    // 15
parseInt("15e2", 10);    // 15
parseInt("15px", 10);    // 15
parseInt("12", 13);      // 15
```

Math

Math nos aporta infinidad de recursos matemáticos avanzados como la *constante de Euler*, gestión de *logaritmos*, *senos*, *cosenos*, *tangentes*… Cada lector debe de indagar y valorar lo que realmente quiere usar, ya que muchos de estos métodos y propiedades van más allá de nuestros objetivos, y no aportan directamente valor al contexto de aprender a programar en JavaScript.

En general, cuando necesitemos desarrollar una aplicación con un gran soporte matemático recurriremos a librerías como *Mathjs*.

Métodos

Hacemos una recopilación simplificada de los métodos más utilizados, aunque existen muchos más.

- ***.round()***

  Devuelve el valor de un número redondeado al entero más cercano.

```
Math.round(20.5);   // 21
Math.round(20.49);  // 20
Math.round(-20.51); // -21
```

- ***.Floor()***

  Devuelve el máximo entero menor o igual a un número.

```
Math.floor(20.5);   // 20
Math.floor(20.49);  // 20
Math.floor(-20.51); // -21
```

- ***.random()***

  Devuelve un número pseudo-aleatorio entre 0 y 1.

  - Número aleatorio entre 0 (incluido) y 1 (excluido).

```
Math.random();
```

  - Número aleatorio entre min (incluido) y max (excluido).

```
Math.random() * (max - min) + min;
Math.random() * (11 - 0) + 0;
```

  - Número entero aleatorio entre min (incluido) y max (excluido).

```
Math.floor(Math.random() * (11 - 0)) + 0;
```

Dates

Sin duda, uno de los elementos que más utilizaremos a lo largo de nuestro camino por el mundo de JavaScript será el manejo de fechas.

Desgraciadamente, en JavaScript el manejo de fechas es bastante complejo y poco intuitivo, lo que ha dado pie, a que existan muchas librerías, que nos facilitan mucho el día a día. Entre ellas:

Dates.js
: Que nos permite una manipulación rápida y precisa con una sintaxis muy intuitiva.

Momments.js
: Que nos aporta mejoras en la sintaxis, además de un complemento genial para gestionar zonas horarias.

Timeago
: Que nos permite convertir fechas a "hace xxx minutos, segundos, días…" de forma dinámica.

Trabajando con fechas

El punto de partida es trabajar *instanciando* a *Date()*, para poder tener nuestra fecha… Para aquellos que estéis familiarizados con la programación orientada a objetos, aquí estamos usando *new* para instanciar una nueva fecha.



Para aquellos que no estéis familiarizados aún, no pasa nada, porque hablaremos de ello más adelante… simplemente recordad, que se utiliza *new* cuando creamos una nueva fecha.

Existen muchas formas de crear nuevas fechas, en función de que parámetro pasamos a *Date(mi_parametro)*.

- **Fecha Actual**

```
var ahora = new Date();
console.log(ahora);
```

- **Usando milisegundos (desde el 1/1/1970 00:00)**

  Muy recomendable.

```
var diaDespues = new Date(3600*24*1000);
console.log(diaDespues);
```

- **Usando cadenas de texto**

  Poco recomendable.

```
var newYear = new Date("January 1, 2016 00:00:00");
```

- **Usando números**

  Recomendable, aunque tiene cierta complejidad oculta a simple vista.

```
var newYear = new Date(2016, 1, 1);          // AAAA, MM, DD
var newYear = new Date(2016, 1, 1, 0, 0, 0); // AAAA, MM, DD, HH, MM, SS
```

- ***Usando UTC***

```
var newYear = new Date(Date.UTC(2016, 1, 1));
```

Métodos

Hacemos una recopilación simplificada de los métodos más utilizados, aunque existen muchos más.

En el caso de las fechas, podemos dividir casi todos los métodos en tres categorías principales:

Getters
: Que nos devuelven información concreta.

Setters
: Que nos permiten ajustar información concreta.

Otros
: Que nos facilitarán enormemente el trabajo.



Importante

Todos aquellos métodos que son susceptibles de sufrir variaciones por la zona horaria, cuentan con un “método clon” que realiza la misma función, pero siguiendo las especificaciones de la *UTC*, como veremos a continuación.

Por otro lado, los meses y los días de la semana, empiezan a contarse desde el 0, y los días de la semana empiezan en domingo siendo este el día 0. El resto de componentes no sufren modificaciones (año, día del mes, etc..).

Getters

- **Versión UTC:**

```
var ahora = new Date();
console.log("Con UTC: ");
console.log("==== FECHA ====");
console.log("El año: " + ahora.getUTCFullYear());         // 4 dígitos
console.log("El mes: " + ahora.getUTCMonth());            // 0 - 11
console.log("El día de la semana: " + ahora.getUTCDay()); // 0 - 6 (D - S)
console.log("El día del mes: " + ahora.getUTCDate());     // 1-31
console.log("==== HORA ====");
console.log("Horas: " + ahora.getUTCHours());
console.log("Minutos: " + ahora.getUTCMinutes());
console.log("Segundos: " + ahora.getUTCSeconds());
console.log("milisegundos: " + ahora.getUTCMilliseconds());
```

- **Versión Local:**

```
var ahora = new Date();
console.log("La fecha es " + ahora);
console.log("==== FECHA ====");
console.log("El año: " + ahora.getFullYear());          // 4 dígitos
console.log("El mes: " + ahora.getMonth());             // 0 - 11
console.log("El día de la semana: " + ahora.getDay());  // 0 - 6 (D - S)
console.log("El día del mes: " + ahora.getDate());      // 1-31
console.log("==== HORA ====");
console.log("Horas: " + ahora.getHours());
console.log("Minutos: " + ahora.getMinutes());
console.log("Segundos: " + ahora.getSeconds());
console.log("Milisegundos desde 1/1/1970: "+ ahora.getTime());
console.log("milisegundos: " + ahora.getMilliseconds());
console.log("==== OTROS ====");
console.log("Diferencia horaria respecto a UTC: " + ahora.getTimezoneOffset());
```

Setters:

Los setters en el caso de las fechas están planteados como una manera de ajustar la fecha y la hora dentro de un marco determinado. Veamos en el siguiente ejemplo:

- **Dentro del marco**

```
var newYear = new Date(Date.UTC(2016, 1, 1));
console.info("La fecha es " + newYear);

newYear.setFullYear(1980);          // Pasamos a 1980
console.info("La fecha es " + newYear);

newYear.setUTCMilliseconds(1500);   // 1500ms en formato UTC
console.info("La fecha es " + newYear);
```

- **Fuera del marco**

  Debes tener en cuenta las leyes naturales: un día tiene 24 horas, un mes tiene un máximo de 31 días…

  Si obviamos esta lógica, JavaScript improvisa una nueva fecha, pero ésta no suele ser correcta.

```
newYear.setDate(56);        // Al día 56 de Enero
console.info("La fecha es " + newYear);

newYear.setUTCHours(36);    // Pasamos a la hora 36 del día
console.info("La fecha es " + newYear);

newYear.setMonth(-6);       // Retrocedemos 6 meses
console.info("La fecha es " + newYear);
```

Otros:

- ***.toString(), .toDateString(), .toTimeString()***

  Devuelve una cadena de texto con la fecha.

```
var ahora = new Date();
ahora.toString();      // Fecha y Hora
ahora.toDateString(); // Solo Fecha
ahora.toTimeString(); // Solo Hora
```

- ***.toUTCString(), .toISOString()***.

  Devuelve una cadena con la fecha en formatos específicos *UTC* e *ISO 8601*.

  El formato *ISO 8601* será de gran ayuda para comunicarnos con otras plataformas, librerías y sistemas, por tanto… **¡aprende como funciona!**

```
var ahora = new Date();
ahora.toISOString();
ahora.toUTCString();
```

- ***.toLocaleString()***

  Devuelve una cadena con la fecha en version local. Si quieres un idioma/región específica deberás recurrir a la *Lista de códigos IETF*.

  También permite una configuración detallada.

```
var ahora = new Date();
console.info(ahora.toLocaleString());

// Código de idioma IETF para Alemán
console.info(ahora.toLocaleString("de-DE"));

// Opciones Avanzadas para fechas
var opciones = {
    weekday: 'long',
    year: 'numeric',
    month: 'long',
    day: 'numeric'};
console.log(ahora.toLocaleString("de-DE", opciones));
```

Benchmark

Ahora que ya sabemos manejar el tiempo, fácilmente podremos hacer pruebas básicas de rendimiento.

Podremos crear una primera fecha antes de ejecutar cierto código y otra justo al final del mismo para así comparar el tiempo transcurrido.

Las pruebas de rendimiento (benchmarks), pueden ser de gran ayuda para tomar decisiones importantes a la hora de refactorizar nuestro código.

```
var inicio = new Date();
    // Código a testear
var fin = new Date();
var transcurso = fin.getTime() - inicio.getTime();
console.info("Pasaron "+transcurso+"ms");
```



Recuerda

Se pueden realizar pruebas de rendimiento con la consola del navegador (*console.time()* y *console.timeEnd()*), como vimos en el capítulo 3.

Setters, problema resuelto



Usa librerías

Lidiar con fechas es una tarea compleja. Si vas a usar librerías, puedes saltarte el resto del capítulo sin ningún problema.

Como vimos antes, en ocasiones deseamos poder sumar 56 días a la fecha actual. Si utilizamos solamente los *setters*, JavaScript estimará fechas seguramente erróneas. Para asegurarnos que estamos sumando y restando las cantidades correctas, lo ideal es siempre incluir el *getter* correspondiente dentro del *setter*.

Veamos a continuación el uso de getters para modificar fechas (días, meses, etc…).

Nota: Partiendo del *ejemplo de MDN*.

Sin *getters*

```
var theBigDay = new Date(1962, 6, 7);
theBigDay.toLocaleString(); // 6/7/1962 23:00:00

var theBigDay = new Date(1962, 6, 7);
theBigDay.setDate(24);
theBigDay.toLocaleString() // 23/7/1962 23:00:00

var theBigDay = new Date(1962, 6, 7);
theBigDay.setDate(32);
theBigDay.toLocaleString() // 31/7/1962 23:00:00

var theBigDay = new Date(1962, 6, 7);
theBigDay.setDate(22);
theBigDay.toLocaleString() // 21/7/1962 23:00:00

var theBigDay = new Date(1962, 6, 7);
theBigDay.setDate(22 + 32 +24);
theBigDay.toLocaleString() // 15/9/1962 23:00:00
```

Con *getters*

```
var theBigDay = new Date(1962, 6, 7);
theBigDay.toLocaleString(); // 6/7/1962 23:00:00

var theBigDay = new Date(1962, 6, 7);
theBigDay.setDate(theBigDay.getDate() + 24);
theBigDay.toLocaleString(); // 30/7/1962 23:00:00

var theBigDay = new Date(1962, 6, 7);
theBigDay.setDate(theBigDay.getDate() + 32);
theBigDay.toLocaleString(); // 7/8/1962 23:00:00

var theBigDay = new Date(1962, 6, 7);
theBigDay.setDate(theBigDay.getDate() + 22);
theBigDay.toLocaleString(); // 28/7/1962 23:00:00

var theBigDay = new Date(1962, 6, 7);
theBigDay.setDate(theBigDay.getDate() + 22 + 32 + 24);
theBigDay.toLocaleString(); // 22/9/1962 23:00:00
```

Capítulo 7 - Cadenas de texto



Uno de los grandes cambios que llegaron con ECMA6 son las *plantillas de cadenas de texto*. Que nos permiten almacenar un texto con espacios, tabulaciones, saltos de líneas… y además nos aporta postprocesado e interpolación de expresiones.

Propiedades

- ***.length()***

  Devuelve la longitud de la cadena, lo que resulta muy importante ya que podremos utilizar bucles para recorrer cada una de las letras…

```
var cadena = "JavaScript, ¡Inspírate!";
console.log("JavaScript, ¡Inspírate! tiene " + cadena.length + " caracteres.");
console.log("Una cadena vacía tiene " + ''.length + " caracteres.");
```

Métodos

Hacemos una recopilación simplificada de los métodos más utilizados, aunque existen muchos más.

- ***.toLowerCase()***

  Devuelve todo en minúsculas.

```
console.log('¡INSPÍRATE!'.toLowerCase());
```

- ***.toUpperCase()***

  Devuelve todo en mayúsculas.

```
console.log('¡inspírate!'.toUpperCase());
```

- ***.fromCharCode()***

  Devuelve una cadena creada mediante una secuencia *Unicode*.

```
  console.info("Es el año 2016 ("+ String.fromCharCode(8559,8559,8553,8548,8544) \
+ ")");
```

- ***.anchor()***

  Crea un *link interno (ancla)* HTML.

```
  document.body.innerHTML = "Contenidos".anchor("contenidos");
  // "<a name="contenidos">Contenidos</a>"
```

- ***.charAt()***

  Devuelve el carácter específico.

```
  var cadena = "JavaScript, ¡Inspírate!";
  console.log("El carácter(posición) 3 es '" + cadena.charAt(3) + "'")
```

- ***.charCodeAt()***

  Devuelve el valor Unicode.

```
  var cadena = "JavaScript, ¡Inspírate!";
  console.log("El carácter(posición) 3 es '" + cadena.charAt(3) + "', en unicode \
("+cadena.charCodeAt(3)+")")
```

- ***.concat()***

  Combina el texto de dos o más cadenas.

```
  var cadena1 = "Oh ";
  var cadena2 = "qué maravillosa ";
  var cadena3 = "mañana'.";
  var combinacion = cadena1.concat(cadena2,cadena3);
  console.log(combinacion); // devuelve "Oh qué maravillosa mañana'."
```

- ***.indexOf()***

  Devuelve la posición del elemento si es capaz de localizarlo o -1 en caso contrario.

```
  "Mundo Web".indexOf("Web") // 6
  "Mundo Web".indexOf("web") // -1
```

- ***.lastIndexOf()***

  Devuelve la ultima localización del elemento si es capaz de encontrarlo o -1 en caso contrario.

```
  "JavaScript, ¡Inspírate!".lastIndexOf("c"); // 16
  "JavaScript, ¡Inspírate!".lastIndexOf("b"); // -1
```

- ***.link()***

  Crea un enlace.

```
var textoActivo="JavaScript, ¡Inspírate!"
var url="https://leanpub.com/javascript-inspirate/"
document.body.innerHTML = "Haga click para volver a " + textoActivo.link(url);
```

- ***.trim()***

  Elimina espacios vacíos al principio y al final de la cadena.

```
console.log('    ¡inspírate! '.trim()); // ¡Inspírate!
```

- ***.slice()***

  Devuelve una cadena nueva con una sección de otra.

```
var cadena = "JavaScript, ¡Inspírate!";
console.log(cadena.slice(12));      // ¡Inspírate!
console.log(cadena.slice(0, 10));   // JavaScript
console.log(cadena.slice(13, -1));  // Inspírate
```

- ***.split()***

  Divide una cadena usando un separador y retorna un array.

```
var cadenaMeses = "Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec";
var meses = cadenaMeses.split(",");
console.log("Los meses son un Array?", Array.isArray(meses))
```

- ***.substr()***

  Devuelve los caracteres de una cadena, comenzando en la localización especificada, y en el número de caracteres especificado.

```
var cadena = "JavaScript, ¡Inspírate!";
console.log("cadena.substr(12, 12):", cadena.substr(12, 12));
console.log("cadena.substr(0,10):", cadena.substr(0, 10));
```

- ***.substring()***

  Devuelve un subconjunto.

```
var cadena = "JavaScript, ¡Inspírate!";
console.log("(0,5): "    + cadena.substring(0,10));
```



Como puedes ver existen muchos métodos similares para retornar partes de una cadena. Sin embargo cada uno tiene matices que los hacen diferentes. Te recomiendo leer *Using Slice(), Substring(), And Substr() in Javascript de Ben Naden*.

Capítulo 8 - Arrays

Los arrays son estructuras que nos permiten almacenar muchos datos, sin tener que preocuparnos por el orden o la organización interna.

Otra forma más sencilla de entenderlo, es imaginar que un array es sencillamente como una lista de la compra.

Partiendo de esta analogía será sencillo añadir elementos a nuestra lista:

```
- Tomates
- Café
- Brócoli
- Cilantro
- Bombillas
```

Como el orden de los elementos en principio nos da igual, utilizaremos un marcador numerico para referirnos a cada elemento de la lista. lógicamente empezaremos por el cero, ya que somos programadores.

```
0 - Tomates
1 - Café
2 - Brócoli
3 - Cilantro
4 - Bombillas
```

Cuando añadimos elementos a la lista el orden puede alterarse o no, en función de si se añadiran antes o después de otros elementos.

```
0 - Tomates
1 - Café
2 - Te (Nuevo)
3 - Brócoli (movido)
4 - Cilantro (movido)
5 - Bombillas (movido)
6 - Pen Drive
```

Cuando eliminamos elementos tambien alteramos la lista…. en función de si están delante de otros elementos o no.

```
0 - Tomates
1 - café
2 - Te
3 - Brócoli
4 Cilantro (Borrado)
4 - Bombillas (movido)
6 - Pen Drive (Borrado)
```

Lógicamente dentro de un array podremos almacenar todo tipo de datos (cadenas, números, objetos, funciones…



En otros lenguajes de programación existen más estructuras similares a estas listas (arrays) como es el caso de las tuplas en Python.

Manejo

Creando un array

Vacío:

```
var arreglo = [];
```

Con elementos:

```
var arreglo = [1, "plátano", "piscina", "manzana", true];
```

Usando el Índice

```
var arreglo = [1, "plátano", "piscina", "manzana", true];
console.log("arreglo[1]:", arreglo[1]);
```

Cambiar un valor del Índice

```
var arreglo = [1, "plátano", "piscina", "manzana", true];
arreglo[0] = "fresa";
arreglo[4] = "pera";
arreglo[2] = "limón";
```

Borrando elementos

Sobreescribiendo a *undefined*

En ocasiones solo queremos dejar el hueco vacío y no cambiar el orden de los elementos de un array.

En estos casos lo mejor es sustituir el valor por *undefined* ya sea usando *delete* o igualando.

```
var arreglo = [1, "plátano"];
arreglo[0] = undefined;
delete arreglo[1];
```

Borrando el elemento

Al eliminar un elemento del array cambiamos el orden dentro del array.

```
var arreglo = [1, "plátano", "manzana"];
arreglo.splice(1, 1);
console.log(arreglo[1]) // manzana
```

Propiedades

- ***.length***

  Podremos saber cuantos elementos contiene un array.

  ```
  var arreglo = [1, "plátano", "manzana"]
  arreglo.length; // 3
  ```

  Al tener este dato clave podremos hacer bucles que realicen sus iteraciones en función de la cantidad de elementos.

  Presta atención al uso de la *variable i* en el interior del bucle para recorrer el array.

```
var numeros = [1, 2, 3, 4, 5];
for (var i = 0; i < numeros.length; i++) {
    numeros[i] *= 10;
}
```



Más adelante veremos que existen estructuras más optimizadas para iterar sobre un array como *map()* y *forEach()*.

Métodos

Hacemos una recopilación simplificada de los métodos más utilizados, aunque existen muchos más.

- ***.isArray()***

  Retorna un booleano en función de si el parámetro es un array o no.

```
var arreglo = [1,2,3]

// Estos son true
Array.isArray([1]);
Array.isArray(arreglo);

// Estos son false
Array.isArray();
Array.isArray({});
Array.isArray(null);
Array.isArray(undefined);
```

- ***.sort()***

  Permite organizar los elementos de un Array, por orden alfabético o en función numérica (ascendente).

```
var frutas = ['plátano', 'Naranja', 'Limón', 'Manzana', 'Mango'];
      frutas.sort(); // ["Limón", "Mango", "Manzana", "Naranja", "plátano"]
      var numeros = [0, 45, 2, -5, 123, -47];
      numeros.sort() // [-47, -5, 0, 123, 2, 45]
```

Aunque en ocasiones el ordenado puede no funcionar como esperamos, si los elementos que componen el Array son de una naturaleza distinta.

```
var miArray = ['uno', 2, true, 'más datos...'];
      miArray.sort(); // [2, "más datos...", true, "uno"]
```

- ***.reverse()***

  Invierte el orden de un array.

```
var miArray = ['uno', 2, true, 'más datos...'];
miArray.reverse();
console.log(miArray) // ["más datos...", true, 2, "uno"]
```

- ***.join()***

  Retorna una cadena con todos los elementos de array dentro.

```
var array = ['dato1', 2, 'masDatos'];
var datosJuntos = array.join();        // 'dato1,2,masDatos'
var datosJuntos2 = array.join('');     // 'dato12masDatos'
var datosJuntos3 = array.join(' + '); // 'dato1 + 2 + masDatos'
```

- ***.toString()***

  Retorna una cadena de texto con todos los elementos.

```
var amigos = ['Luis', 'Carlos', 'Marco', 'Eduardo'];
console.log(amigos.toString());
```

- ***.toLocaleString()***

  Retorna como string (configuración regional) todos los elementos.

```
var numero = 1337.89;
var fecha = new Date();
var miArray = [numero, fecha, 'más datos'];

var arrayConvertida = miArray.toLocaleString();
console.log(arrayConvertida);
```

- ***.concat()***

  Retorna un nuevo array con los arrays especificados concatenados.

  - Dos arrays:

```
var arreglo = ['a', 2, true];
      var arreglo2 = [1, 2, 4];

      var nuevaArray = arreglo.concat(arreglo2);

  console.log(nuevaArray);
```

- Múltiples arrays:

```
var arreglo = ['a', 2, true];
    var arreglo2 = [1, 2, 4];
    var otroArreglo = ['abc', 1, false]

    var nuevaArray = arreglo.concat(arreglo2, [5.25, 100], otroArreglo);

  console.log(nuevaArray);
```

- ***.indexOf()***

  Devuelve la posición donde se encuentra el elemento en sí ó -1, si no lo encuentra.

```
var array = [2, 5, 9];
    var index = array.indexOf(9);  // 2
    var index = array.indexOf(12); // -1
```

- ***.lastIdexOf()***

  Devuelve la posición del último elemento en sí que coincide ó -1, si no lo encuentra.

```
var array = [7, 1, 3, 7];
array.lastIndexOf(7); // 3
array.lastIndexOf(2); // -1
```

- ***.push()***

  Añadir nuevos elementos al final de un array.

```
var arreglo = [1, "plátano", "manzana"];
console.log("Antes:", arreglo.length);
arreglo.push("nuevo");
console.log("Después:", arreglo.length);
console.log("arreglo[4]:", arreglo[4]);
```

- ***.unShift()***

  Añade nuevos elementos al principio del array.

```
var miArray = [1, 2];
miArray.unshift(true, "otros datos...");
console.log("Longitud actual:", miArray.length);
```

- ***.pop()***

  Eliminar el último elemento del array.

```
var miArray = [1, 2];
arreglo.pop();
console.log("Longitud actual:", miArray.length);
```

- ***.shift()***

  Eliminar el primer elemento del array.

```
arreglo.shift();
```

- ***.splice()***

  Borrar elementos del array, alterando con ello la posición de los demás.

```
var frutas = ['plátano', 'Naranja', 'Limón', 'Manzana', 'Mango'];
console.log("frutas[1]:", frutas[1]) // Naranja
frutas.splice(1, 3);
console.log("frutas[1]:", frutas[1]) // Mango
console.info("frutas.length:", frutas.length); // 2
```

  Si deseamos conservar en un variable aquellos elementos que hemos eliminado, es necesario hacer una asignación como ésta:

```
var frutas = ['plátano', 'Naranja', 'Limón', 'Manzana', 'Mango'];
var citricos = frutas.splice(1, 2);
console.info("citricos:", citricos);
console.info("frutas.length:", frutas.length);
```

Métodos Avanzados

Algunos de los métodos más utilizados, requieren comprender en profundidad el manejo de funciones y en especial el *retorno*.



Como aún no hemos hablado de funciones directamente, nuestra recomendación es leer por encima los siguientes métodos y su funcionamiento… para regresar más adelante a este capítulo.

- ***.some()***

  Verifica si alguno de los elementos del array pasan la prueba implementada por la función dada.

```
function tamañoValido(elemento, indice, arreglo) {
  return elemento >= 10;
}
[12, 5, 8, 130, 44].some(tamañoValido);   // true
[12, 54, 18, 130, 44].some(tamañoValido); // true
```

- ***.every()***

  Verifica si todos los elementos del array pasan la prueba implementada por la función dada.

```
function tamañoValido(elemento, indice, arreglo) {
  return elemento >= 10;
}
[12, 5, 8, 130, 44].every(tamañoValido);   // false
[12, 54, 18, 130, 44].every(tamañoValido); // true
```

- ***.filter()***

  Crea un nuevo array con aquellos elementos que cumplan la condición.

```
function tamañoValido(elemento) {
  return elemento >= 10;
}
var filtrados = [true, 134, 10, 0, null, "Hola"].filter(tamañoValido);
```

- ***.forEach()***

  Se ejecuta la función por cada elemento del array.

```
function logger(element, index, array) {
    console.log("array[" + index + "] = " + element);
}
[2, 5, 9].forEach(logger);
```

- ***.map()***

  Itera sobre el array aplicando una transformación, que definimos en una función y finalmente retorna un nuevo array con todos los componentes modificados.

```
var arreglo = ["plátano", "fresa", "lima", "manzana"];
var resultado = arreglo.map(function (elemento){return elemento + " modificado!\
"});
console.log(resultado);
```

Arrays multidimensionales

No dejéis que el nombre os asuste, sencillamente cuando tenemos arrays almacenadas dentro de otros arrays, entendemos que se trabaja sobre *Matrices o Arrays multidimensionales*. Es sencillo:

```
var arreglo1 = ["plátano", "fresa", "lima", "manzana"];
var arreglo2 = ["entrante", "primero", "segundo", "postre"];

var juntandoArreglos = [arreglo1, arreglo2];

console.log(juntandoArreglos[0][0]); // plátano
console.log(juntandoArreglos[1][3]); // postre
```

Capítulo 9 - Objetos

Objetos Literales

Junto a las funciones, de las cuales hablaremos en el próximo capítulo, los objetos son la clave para dominar el JavaScript de hoy.



Pero antes de nada aclaremos un concepto básico:

No es necesario saber *programación orientada a objetos*, para trabajar con objetos.

De hecho hasta la llegada de ECMAScript 6 no había algo como *Class*, ya que en JavaScript este paradigma se hace de otra forma, gracias a *Prototype*.



Pero… ¡no os agobiéis! con saber que no estamos haciendo *Programación Orientada a Objetos (POO)* nos vale.

Vamos a trabajar con *Objetos (Objetos literales)*, básicamente son como arrays, en el sentido que son una variable especial que nos permite almacenar muchas cosas dentro de si misma.

A diferencia de las arrays, aquí las cosas tienen nombre (propiedades) y en caso de almacenar una función dentro del objeto recibirá el nombre de *Método*, aunque sea exactamente igual que una función normal.

La forma de almacenar datos dentro de objetos, tiene mucha relevancia, y por eso la mayoría de servicios web basados en *API REST*, dan soporte al formato de intercambio de datos *JSON (JavaScript Object Notation)*.

JSON básicamente es un objeto de los que veremos a continuación, pero que fue serializado de tal forma que es una cadena de texto, que puede ser parseada de vuelta. Esta forma de trabajar es claramente más fácil que el clásico y obsoleto XML, para el intercambio de datos cliente-servidor en el contexto del desarrollo web.

Manejo

- **Creando un objeto**

  Vacío:

```
var miObjeto = {};
```

  Con propiedades:

```
var miObjeto = {
    cadena: 'esto es una cadena',
    numero: 2,
    booleano: false
};
```

  Con Métodos:

```
var miObjeto = {
    saludar: function(){
        console.log("hola!");
    }
};
```

- **Usando las propiedades y métodos**

```
var miObjeto = {
  metodo: function() {
          console.log(miObjeto.propiedad1)
  },
  propiedad1: "Datos"
};

// Recuperando el valor de propiedad1
console.log("miObjeto.propiedad1:", miObjeto.propiedad1);

// Ejecutando el método
miObjeto.metodo()
```

- **Cambiar un valor de una propiedad**

```
var miObjeto = {
  propiedad1: "Datos"
};

miObjeto.propiedad1 = 123456789;

console.log(miObjeto.propiedad1);
```

- **Borrando elementos**

  Utilizaremos el controvertido operador *delete*. Te recomiendo *esta lectura*, para mejorar tu comprensión acerca de este operador.

```
var miObjeto = {
  propiedad1: "Datos",
  borrame: "Quiero ser borrado"
};

console.log(miObjeto.borrame);

delete miObjeto.borrame;

console.log(miObjeto.borrame);
```

Métodos

Hacemos una recopilación simplificada de los métodos más utilizados, aunque existen muchos más.

- ***.defineProperties()***

  Define nuevas propiedades o modifica las existentes directamente en el objeto, retornando el objeto modificado.



Realmente nos permite llevar nuestros objetos mucho más allá. No obstante trasciende de nuestros objetivos, pero es un buen punto de partida.

Más información…

```
var miObjeto = {propiedad: "Propiedad original..."}
Object.defineProperties(miObjeto, {
  "propiedad1": {
    writable: true,
    value: "Propiedad1 original..."
  },
  "propiedad2": {
    value: "Cadena de texto",
    writable: false
  }
});
console.info(miObjeto);
miObjeto.propiedad1 = "Propiedad1 original Modificada";
console.info(miObjeto.propiedad1);
miObjeto.propiedad2 = "Cadena de texto... ¿modificada?";
console.info(miObjeto.propiedad2);
```

- ***.getOwnPropertyDescriptor()***

  Devuelve los detalles de las propiedades y métodos del objeto. En caso de no existir retornará *undefined*.

```
var miObjeto = {
  metodo: function() {
          console.log(miObjeto.propiedad1)
  },
  propiedad1: "Datos"
};

console.info(Object.getOwnPropertyDescriptor(miObjeto, 'propiedad1'));
// {value: "Datos", writable: true, enumerable: true, configurable: true}

console.info(Object.getOwnPropertyDescriptor(miObjeto, 'inventado'));
// undefined
```

- ***.getOwnPropertyNames()***

  Devuelve un array con todos los nombres de las propiedades y métodos del objeto.

```
var miObjeto = {
  metodo: function() {
          console.log(miObjeto.propiedad1)
  },
  propiedad1: "Datos"
};

console.log(Object.getOwnPropertyNames(miObjeto));
// ["metodo", "propiedad1"]
```

- ***.isExtensible()***

  Determina si un objeto es extensible, es decir que se puedan agregar nuevas propiedades al mismo.

```
var miObjeto = {
  metodo: function() {
          console.log(miObjeto.propiedad1)
  },
  propiedad1: "Datos"
};

console.log("¿Se puede extender?", Object.isExtensible(miObjeto));

var sellado = Object.seal(miObjeto);
console.log("¿Se puede extender?", Object.isExtensible(sellado));

var congelado = Object.freeze(miObjeto);
console.log("¿Se puede extender?", Object.isExtensible(congelado));

Object.preventExtensions(miObjeto);
console.log("¿Se puede extender?", Object.isExtensible(miObjeto));
```

- ***.hasOwnProperty()***

  Devuelve true o false si la propiedad existe o no.

```
var miObjeto = {
  metodo: function() {
          console.log(miObjeto.propiedad1)
  },
  propiedad1: "Datos"
};

console.log("¿Tiene la propiedad \"propiedad1\"?", miObjeto.hasOwnProperty('pro\
piedad1'));
console.log("¿Tiene la propiedad \"propiedad2\"?", miObjeto.hasOwnProperty('pro\
piedad2'));
```

- ***.propertyIsEnumerable()***

  Devuelve true o false, si la propiedad especificada es enumerable, y con ello sabemos si será incluida en la iteración de un bucle *For… In*.

```
  var miObjeto = {
    metodo: function() {
            console.log(miObjeto.propiedad1)
    },
    propiedad1: "Datos"
  };

  console.log("¿Es enumerable \"propiedad1\"?", miObjeto.propertyIsEnumerable('pr\
opiedad1'));
  console.log("¿Es enumerable \"metodo\"?", miObjeto.propertyIsEnumerable('propie\
dad2'));
```

- ***.toLocaleString()***

  Retorna como string (configuración regional) todas las propiedades.

```
  var fecha = new Date();

  var miObjeto = {
    metodo: function() {
            console.log(miObjeto.propiedad1)
    },
    propiedad1: "Datos",
    fecha: fecha
  };

  miObjeto.toLocaleString()
  console.log("La fecha es ", miObjeto.fecha);
```

Métodos Avanzados

Algunos de los métodos más utilizados requieren comprender en profundidad el manejo de funciones y en especial el *retorno*.



Como aún no hemos hablado de funciones, mi recomendación es leer por encima los siguientes métodos y su funcionamiento… para retornar más adelante a este capítulo.

- ***For… in***

  Itera sobre todas las propiedades de un objeto, en un orden arbitrario.

```
var objeto1 = {
    propiedad1: "hola",
    propiedad2: 2,
    propiedad3: false,
    propiedad4: [true,2,5, "..."],
    propiedad5: {
        dato: "más datos..."
    },
    metodo: function(){
        console.log("hola");
    }
}

function mostrar_propiedades(objeto, nombreObjeto) {
   var resultado = "";
   for (var i in objeto) {
      resultado += nombreObjeto + "." + i + " = " + objeto[i] + "\n";
   }
   return resultado;
}

mostrar_propiedades(objeto1, "objeto1");
```

Usos Especiales

Trabajando con espacios y caracteres especiales

```
var miObjeto = {
        nombre: "objeto",
        "año": 2015,
        "estado del sistema": "correcto"
};

console.log(miObjeto["año"]);
miObjeto["estado del sistema"] = "fuera de servicio";
console.log(miObjeto["estado del sistema"]);
```

Acortar objetos

En ocasiones, especialmente al parsear datos de internet en formato JSON, los objetos pueden estar muy anidados… dificultando enormemente el manejo. Por eso podremos considerar una buena práctica, acortarlos con nuevas variables o sobreescribir al propio objeto con aquellos datos relevantes:

```
var objetoAbreviado = objeto.muy.largo["metodos y propiedades"];

objetoAbreviado.propiedad1;
```

Estructuras de datos

Un buen manejo de objetos, nos dará las claves para poder tener una estructura de datos eficiente en nuestra aplicación y en las *comunicaciones cliente-servidor*.



Los objetos, sientan las bases sobre las que luego almacenaremos mucha más información, en el capítulo 12, veremos mucho más sobre esto.

Capítulo 10 - Funciones

Las funciones de JavaScript son el alma de este lenguaje, por eso se consideran *ciudadanos de primera clase (first-class citizen)*, además de *entidades de orden superior*.

En JavaScript, las funciones tienen *"super poderes"*. Estos son algunos de los más importantes:

- Ser pasadas como parámetros *(callbacks)*.
- Ser parte de los objetos como métodos.
- Ser asignadas a una variable (función anónima).
- Ser retornadas por otra función.

Una de las claves para entender la importancia de las funciones, aún cuando estamos dando nuestros primeros pasos en JavaScript es la reusabilidad. Podemos crear partes de código que fácilmente podremos reutilizar a lo largo de una aplicación o incluso a lo largo de muchos programas y aplicaciones… llegando incluso a crear nuestras propias librerías.

Pero para dominar la reusabilidad y respetar con profundidad el principio de programación *DRY (Don't Repeat Yourself)*, deberemos en cualquier caso ser capaces de manejar los parámetros y el retorno de las funciones, algo de lo que hablaremos mucho en este capítulo.

Las funciones, especialmente como *parámetro (callback)*, también será nuestra puerta de entrada al maravilloso, caótico y paradigmático *mundo de la asincronía*.

Manejo

- **Declarar funciones**

  Como sentencia:

```
function miFuncion (){
	console.log("Hola!")
}
```

  Como valor de una variable:

```
var miFuncion = function(){
	console.log("Hola!")
}
```

Como método en un objeto:

```
var miObjeto = {
        propiedad: "Soy una propiedad",
        metodo: function(){
                console.log("Hola!")
        }
}
```

- **Ejecutar funciones**

  Aunque pueda parecer algo extraño, desde el principio, ya estábamos ejecutando funciones.

```
// Recuerdas isNaN?
console.log("Recuerdas isNaN?", isNaN(NaN))
```

  Ahora ejecutamos nuestras propias funciones y métodos.

```
var miFuncion = function(){
    console.log("Hola!");
}

function otraFunción() {
    console.log("Hola de nuevo!");
}

var obj = {
    metodo: function () {
        console.log("Hola... ahora como método!");
    }
}

miFuncion();
otraFunción();
obj.metodo();
```

Argumentos y parámetros

Cuando queremos hacer funciones con un nivel de abstracción realmente alto, tenemos que recurrir al aislamiento. De tal forma que nuestra función no dependa de ciertas variables o datos externos a ella.

Cuando definimos (creamos) una función, podemos incluir ciertos parámetros entre los paréntesis que actuarán como referencias. Funcionarán internamente igual que variables, de tal forma que a la hora de ejecutar la función… podremos pasarle ciertos argumentos y así tener funciones con un mayor nivel de abstracción.

Uso Normal

```
// Declarando Parámetros
function sumar (p1, p2){
        console.log("suma:", p1 + p2)
}

// Pasando Argumentos
sumar(2, 3);
```

El exceso de argumentos no es un problema.

```
// Declarando Parámetros
function sumar (p1, p2){
        console.log("suma:", p1 + p2)
}

// Pasando Argumentos
sumar(2, 3, "más datos...", 45, true);
```

La falta de argumento crea un valor indefinido.

```
function testeando (p1, p2){
        console.log("p1:", p1);
        console.log("p2:", p2)
}

// Pasando Argumentos
testeando(2);
```

Parámetros opcionales

Podremos simplificar enormemente la ejecución de las funciones si, definimos ciertos valores por defecto para aquellos parámetros que consideremos opcionales.

Este trabajo adicional por nuestra parte, se verá recompensado posteriormente en tareas de soporte y documentación que no tendremos que realizar.

Trabajar con valores por defecto nos ayudará mucho para construir librerías y un código modular eficiente.

Básicamente existen dos maneras de hacer esto.

- **Utilizando el operador ||**

```
function userID(nombre, numero) {
  numero = numero || "000000E";
  console.log("ID:", nombre + "-" + numero)
}

userID("Ulises", 31); // Ulises-31
userID("Oscar");      // Oscar-000000E
userID("Pepe", 0)     // Pepe-000000E
```

Aunque este operador hace un buen trabajo se equivoca con el 0 -entre otros- por eso no es recomendable utilizarlo, especialmente cuando se encarga de gestionar el parámetro por defecto de valores numéricos.

- **Utilizando un *if***

Podemos hacer una validación por tipo, lo que descartará ciertos falsos positivos como en el caso del 0.

```
function sumar(a, b) {
  if(typeof b === 'undefined'){
          b = 0;
  }

  return a+b;
}

sumar(2);    // 2
sumar(2, 8); // 10
```

Con un operador ternario se hace más compacto pero menos legible:

```
function sumar(a, b) {
  b = typeof b !== 'undefined' ?  b : 0;
  return a+b;
}

sumar(2);    // 2
sumar(2, 8); // 10
```

El orden es clave

El orden de los parámetros es muy importante, ya que su posición puede alterar enormemente la usabilidad a la hora de la ejecución, por eso el orden siempre será:

- Parámetros fijos (primero).
- Parámetros opcionales (después).

Objetos como argumento

Se considera una buena práctica, pasar un único objeto como parámetro si estamos manejando más de tres parámetros fijos.

De esta forma además de agrupar todo fácilmente, también podemos cambiar el orden de entrada de datos.

Es importante recordar que debemos documentar muy bien lo que esperamos, que contenga el objeto, de lo contrario nuestros métodos y funciones pueden ser un infierno para cualquier otro programador e incluso para nosotros mismos pasado un tiempo.

```
contactos = [];

function crearContacto (nombre, usuarioTwitter, referencias, notas, fotoUrl){
        contactos.push({
                "nombre": nombre,
                "@": "@" + twitter
        })
}

crearContacto("Oscar", "inventado", "amigos...", "etc...", "más cosas...");
```

¡Refactorizemos!

```
contactos = [];

function crearContacto (datos){
        contactos.push({
                "nombre": datos.nombre,
                "@": "@" + datos.twitter
        })
}

// Puedo pasar los atributos en el orden que quiera
crearContacto({twitter: "inventado", nombre: "Pepe", fotoUrl: "http..."});
```

Avanzado: Objeto *arguments*

El Objeto Arguments no es un array, solo es similar.

```
function pruebaArgumentos () {
        console.log(arguments);
        console.info(arguments[0]);
        console.info(arguments[1]);
}

pruebaArgumentos (1, "vale", true);
```



Conversión array requiere de ciertos conocimientos avanzados en el uso de *prototype* y *this*. Os dejo una función que os ayudará a realizar esta conversión de una forma fácil.

```
function conversorArgumentos(arguments) {
    var argumentos = Array.prototype.slice.call(arguments);
    return argumentos.sort();
}
```

Retorno

Otro de los puntos fuertes a la hora de plantear estructuras de código modulares y reutilizables, es tener en cuenta el retorno.

El retorno nos permite devolver un valor al terminar de ejecutarse la función. Este valor puede ser cualquier tipo de dato de los muchos que tenemos en JavaScript. Por supuesto, también funciones y objetos.

Cómo utilizar funciones que retornen valores en función de ciertas operaciones realizadas.

```
        function validarPar(numero){
                var esPar = numero % 2 !== 1;
                var mensaje;

                if (esPar) {
                        mensaje = "Bravo! es un número par!";
                } else {
                        mensaje = "ERROR! No es un número par.... ¬¬\"";
                }
                return mensaje;
```

```
        };

        console.log("El 5 es un número par?", validarPar(5));
        console.log("El 2 es un número par?", validarPar(2));
```

Una suma de cuadrados en el retorno. Las operaciones también pueden ser realizadas en el retorno de la función.

```
function sumaCuadrados (a, b) {
        return (a*a) + (b*b);
};

var resultado = sumaCuadrados(2, 3);
console.log("2x2 + 3x3 =", resultado)
```

Anidación

Dentro de una función, podemos crear nuevas funciones al igual que variables de todo tipo. Este es un recurso a tener en cuenta, pero no debemos abusar de la anidación… ya que, el código puede volverse muy difícil de leer y depurar.

```
function saludar(quien){
        function alertaSaludo(){
                console.log("hola " +  quien);
        }
        return alertaSaludo;
}

var saluda = saludar("Amigo/a");
saluda();
```

También podemos usar parámetros, al igual que una función normal.

```
function saludar(quien){
        function alertaSaludo(){
                console.log("hola " +  quien);
        }
        return alertaSaludo;
}

saludar("Amigo/a")();
```

Ámbito (Scope)

Por defecto en JavaScript existen dos tipos de ámbitos, local y global. Dominar los ámbitos nos hará llegar a ser grandes artesanos, pero no es una tarea sencilla.

En principio aquellas variables que se han declarado fuera de la función, son de ámbito global, y las variables que se declaran en el interior serán consideradas de ámbito local.

Desde cualquier función siempre podremos acceder a todas las variables que se han declarado en el ámbito global, pero desde el exterior de una función no podremos acceder a su ámbito local. Para poder solventar esta limitación se utilizan los retornos que vimos anteriormente y algunos recursos adicionales que veremos más adelante.

```
var ambitoGlobal = "Soy una variable Global!";

function miFuncion () {
        var ambitoLocal = "Soy una variable Local!";
        console.log("Desde -local- puedo ver ambitoLocal?", ambitoLocal);
        console.log("Desde -local- puedo ver ambitoGlobal?", ambitoGlobal);
}

console.log("Desde -global- puedo ver ambitoLocal?", ambitoLocal);
//Uncaught ReferenceError: ambitoLocal is not defined(…)

console.log("Desde -global- puedo ver ambitoGlobal?", ambitoGlobal);
```

Este juego de ámbito local y global, puede extenderse en el entorno compartido y aislado de las funciones anidadas.



Duplicando Variables

Una mala práctica a la hora de planificar nombres de las variables en nuestra aplicación puede llevarnos a la situación en la que tengamos variables creadas (declaradas) en el ámbito global y en el local con los mismos nombres.

Esto puede ser evitado desde la planificación en una fase temprana o posterior con algún *linter* como *JSHint* o *ESLint*.

Funciones Anónimas

En JavaScript podemos crear tantas funciones como queramos, sin embargo entre los requisitos de creación no está incluir un nombre necesariamente.

Funciones que retornan funciones

Cuando una función retorna una nueva función, esta nueva función lógicamente será anónima.

```
function saludo(quien){
    return function(){
        console.log("hola " + quien);
    }
}

var saluda = saludo("Amigo/a");
saluda();
```

Podemos ejecutar ambas funciones, sin asignar una variable necesariamente.

```
function saludo(quien){
    return function(){
        console.log("hola " + quien);
    }
}

saludo("Amigo/a")();
```

Funciones anónimas autoejecutadas



Es uno de los patrones más clásicos y utilizados en JavaScript, para encapsular nuestro código y prevenir que pueda ser alterado desde el exterior.

Esta técnica da mucho juego, si tenemos en cuenta que podemos usar el retorno.

Al aislar nuestro código tanto del exterior, podemos pensar que nuestro programa se queda lejos de ser capaz de interactuar con el usuario, pero esto es incorrecto, ya que en JavaScript podremos recurrir a la *programación dirigida por eventos*. Hablaremos en próximos capítulos sobre ello.

```
(function() {
        console.log("hola Amigo/a")
})();
```

Resulta más sencillo de entender esta estructura si entendemos el juego de los paréntesis.

Declaramos una función:

```
(//código)()
```

Lo contenido en el primer paréntesis contiene el código encapsulado, al igual que hacíamos con las operaciones matemáticas en capítulos anteriores.

El segundo paréntesis es el encargado de ejecutar el bloque de código anterior, asi es como logramos que la función sea inmediatamente ejecutada dentro de un ámbito al que no podremos acceder.

Como podemos ver, la estructura básica sería algo así:

```
(function(){})();
```

Aunque existen bastantes variantes y debates:

```
(function(){}());
!function(){}();
+function(){}();
!1%-+~function(){}();
//...
```

Al igual que el resto de funciones podemos hacer uso de los parámetros.

```
( function(quien){
   console.log("hola " + quien);
})("Amigo/a");
```



Objeto Window como parámetro

Aunque por temas de rendimiento -lo más habitual- es pasar como argumento el *objeto window*, así disponemos de una copia dentro del propio ámbito de la función.

```
(function(window){
  // código
})(window);
```

Recursión

Otra manera más *funcional* y divertida de hacer bucles es utilizando la recursión. Básicamente una función es capaz de llamarse a sí misma durante su ejecución, lo que resulta ser una funcionalidad muy atractíva para ciertas operaciones.



Por otro lado, aunque es una práctica muy habitual entre los programadores -que defienden- la *programación funcional* en JavaScript, puede ser complicado prevenir el riesgo de caer en bucles infinitos.

Un clásico donde podemos aplicar recursividad es en el cálculo del *factorial*.

```
function factorial(n){
        if(n <= 1){
            return 1
          } else {
            return n * factorial(n-1)
        }
}

factorial(0); // n! = 1
factorial(1); // n! = 1
factorial(2); // n! = 2
factorial(3); // n! = 6 (3*2*1)
factorial(4); // n! = 24 (4*3*2*1)
factorial(5); // n! = 120 (5*4*3*2*1)
factorial(6); // n! = 720 (...)
```

Callbacks



La primera curiosidad sobre los callbacks:

Es una técnica de programación y no una facilidad del lenguaje, por ello *callback* no es una palabra reservada en JavaScript, y puedes usarla en tu código, si te resulta más legible.



***Callbacks en Wikiwand*:**

"En programación de computadoras, una devolución de llamada o retrollamada (en inglés: callback) es una función "A" que se usa como argumento de otra función "B". Cuando se llama a "B", ésta ejecuta "A". Para conseguirlo, usualmente lo que se pasa a "B" es el puntero a "A"."

Esto quiere decir, **que cuando cierta función termina de realizar todo lo que tiene que hacer, ejecutará una función que le fue pasada como argumento**.

En un principio, este concepto parece complicado, y sin duda lo es, pero este sistema es el primer paso para manejar la asincronía. Esto sucederá cuando nuestro código deja de ejecutarse de manera estructurada línea a línea, por ejemplo con las *peticiones AJAX*, lo que veremos en próximos capítulos.

Comparando por contexto

Cuando tenemos un código síncrono, fácilmente podemos obviar el uso de callbacks, y llegar al mismo resultado, ya que nuestro código sigue un orden lógico.

Sin Callbacks:

```
function primerPaso() {
        console.log("Este es el primer paso");
};

function segundoPaso() {
        console.log("Este es el segundo paso");
};

primerPaso();
segundoPaso();
```

Con Callbacks:

```
function primerPaso(callback) {
        console.log("Este es el primer paso");
        callback();
};

function segundoPaso() {
        console.log("Este es el segundo paso");
};

primerPaso(segundoPaso);
```

Cuando nuestro código se ejecute de forma asíncrona, la única forma de conservar el flujo en orden, será utilizando entre otras cosas *Callabcks* o *Promesas*, como veremos a continuación.



Si has desarrollado alguna vez con JQuery, habrás notado que tiene unas características ligeramente diferentes al JavaScript al que estamos acostumbrados.

```
$('#elemento').fadeIn('slow', function() {
   // código del callback
});
```

Como puedes ver… en muchos métodos, pasamos como argumento una función que declaramos en línea. Básicamente… **¡ya estábamos usando callabacks!** pero no eramos conscientes.

Veamos un ejemplo, un poco condensado. Os ayudaré comentando el código:

```
/*
        Declaramos una función que espera dos parámetros
        - parametro
        - callback
*/
var quieroCallback = function(p1, callback){
    // Consideramos el callback como algo opcional.
    if (callback){
            // Validamos si es una función o no.
            if (typeof callback === 'function'){
                    /*
                      De ser una función lo ejecutamos y
                      y pasamos como argumento "p1"
                    */
                callback(p1);
            } else {
                    /*
                      Si no se trata de una función...
                      simplemente mostramos ambos datos.
                     */
                console.log(p1, callback);
                }
    }
}

quieroCallback('a', 'b');

quieroCallback('a', function(val){
    console.log(val);
});
```

Asincronía



La naturaleza de la Asincronía

Hasta ahora todo el código que vimos se ejecutaba de una manera lógica, previsible y secuencial. Cada línea de código era ejecutada después de la anterior, tardará lo que tardará. Este estilo de programación es ineficiente y bloqueante, lo que en el mundo de la web es intolerable.

La asincronía es una caracteristica propia de ciertos métodos que permiten su ejecución en un segundo plano. De tal forma que resulta imposible saber cuando terminarán y además antes de terminar su ejecucción se ejecutan la siguiente línea de código.

Cuando en JavaScript se habla de asincronía, lo que realmente está ocurriendo es que dejamos de ejecutar partes de nuestro script de manera secuencial. Esto crea un efecto curioso que tiene como consecuencia, un script muy escalable y rápido, ya que el sistema no espera a que algo termine para seguir ejecutando el resto del script.



La mala noticia, es que recaerá en el lector todo el peso de controlar esos caballos desbocados. La asincronía es tan potente, que no existe otra forma de trabajar sobre Node.js. Por eso Node.js está concebido -de principio a fin- como un sistema asíncrono.

Existen muchas formas de manejar la asincronía.

- Paso de continuadores (Callbacks).
- Eventos.
- Promesas (ECMA6 y librerías…).
- Generadores (ECMA6, Closures, etc…).

Nosotros veremos en este capítulo -exclusivamente- la gestión de asincronía por medio de callbacks.

En el próximo capítulo hablaremos de *programación dirigida por eventos* y como gestionar con ello la asincronía.

Para hacer un poco más fluido esta explicación, utilizaremos *setTimeout* que por defecto es una función asíncrona.

Veamos como funciona el código sin gestionar la asincronía:

```
function traigoDatos (){
    // Asincrona
    setTimeout (function(){
        console.log ("Esto son mis datos");
    },2000)
}

function pintoDatos(){
    // No asincrona
    console.log("ya tengo los datos");
}
traigoDatos();
pintoDatos();
```

Como puedes ver… los mensajes no salen en el orden correcto. Recuerda que, para pintar datos, el paso previo -siempre- es tener esos datos disponibles.

Ahora vamos a intentar resolver este problema de una manera sencilla. Si introducimos un callback en la función asíncrona, seremos capaces de resolver el problema… aunque tarde 3 segundos o 5 minutos.

```
function traigoDatos (callback){
    // Asincrona
    setTimeout (function(){
        console.log ("Esto son mis datos");
        // Llamamos a Callback cuando haya llegado el fin de traigoDatos.
        callback();
    },2000)
}

function pintoDatos(){
    // No asincrona
    console.log("ya tengo los datos");
}

traigoDatos(pintoDatos);
```

Al ejecutarlo podemos ver que el problema de la asincronía ha sido resuelto.

Normalmente, a la hora de hacer peticiones asíncronas, solemos *pedir/enviar* información al servidor… y hacemos esto a través de peticiones AJAX (también asíncronas). Cuando realizamos ese tipo de llamadas, queremos pasarle al callback los datos que nos han llegado del servidor.

¡Veamos como hacerlo!

```
function traigoDatos (callback){
    // Asíncrona
    setTimeout (function(){
        // muchas cosas pasan...
        var resultado = "Esto son mis datos";
        // Llamamos a Callback y pasamos el resultado
        callback(resultado);
    },2000)
}

function pintoDatos(data){
    // No asíncrona
    console.log("ya tengo los datos:");
    console.log(data);
}

traigoDatos(pintoDatos);
```

Sobrevivir al *Callback Hell*

Callback Hell es una situación que se suele producir cuando los programadores no dominan el manejo de la asincronía, ni el uso de los callbacks. También se produce, cuando no han respetado conceptos básicos de modularización y prevención de anidación desmedida.

Algunas soluciones a este problema:

- No anidar en exceso… ¿Has oído hablar de la *complejidad ciclomática*?.
- Cualquier anidación de funciones a más de dos o tres niveles está pidiendo a gritos una refactorización.
- No todas las funciones de tu código han de ser anónimas…
- Modularizar y refactorizar son tus dos mejores amigos en JavaScript.
- Gestiona los errores en cada función y no al final de la pila.

Si aún así te ves totalmente incapaz de prevenir este error, siempre puedes recurrir a *Generadores*, *Promesas*, *Funciones Async*… o librerías como *Async*, *Q*, etc…

Documentar

Si recordamos el tercer capítulo, dijimos que *JSDoc* nos resultaría muy útil en el futuro para entender y documentar especialmente nuestras funciones. Veamos de nuevo aquel ejemplo, esta vez con una mirada más crítica.

```
/**
 * Retorna los detalles del libro.
 * @param {string} title - Título del libro.
 * @param {string} author - Autor del libro.
 * @returns {object} title, author, picture (referencia local), code
 */
function Book(title, author) {
        return {
                title: title,
                author: author,
                picture: "../images/"+author+"/"+title+".jpg",
                code: 010203 + author + "/" + title
        }
}
```



¡Volver atrás!:

Ahora puede ser un buen momento para volver a capítulos anteriores, donde era necesario hacer uso de las funciones para gestionar ciertos métodos complejos en arrays y objetos.

Parte III - Web dinámica y conectada...

Capítulo 11 - Hackeando HTML y CSS

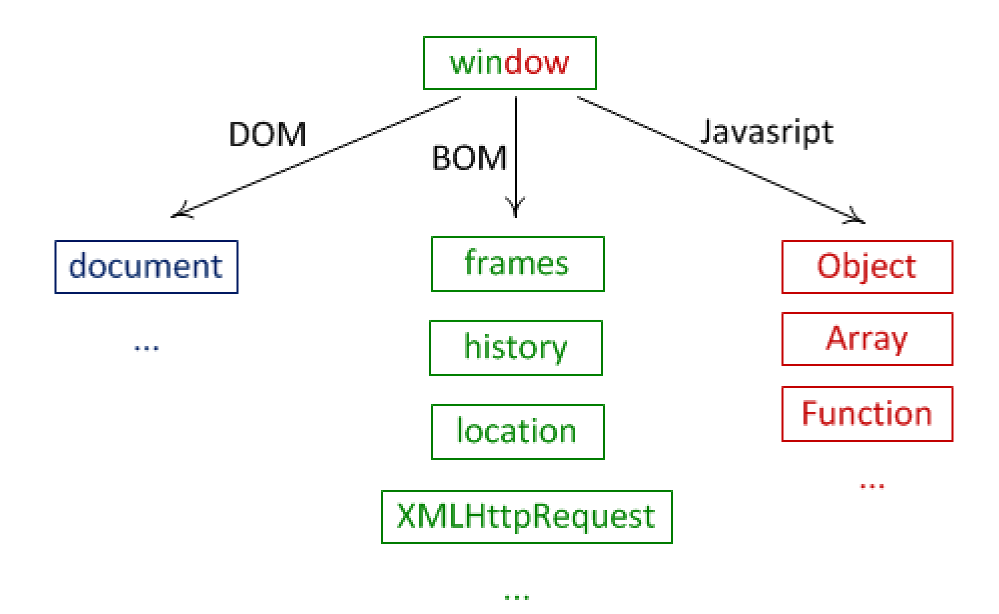


Imagen obtenida de javascript.info

A la hora de trabajar con el navegador fuera de la consola, disponemos básicamente de tres pilares:

JavaScript
: Que es lo que intentamos aprender a lo largo de este libro, y que nos permite en esencia programar sobre la web que está renderizada en el cliente.

BOM
: *Browser Object Model*, que contiene navigator, *history*, *screen*, *location*, *XMLHttpRequest*, etc… los cuales son hijos de *window*.

DOM
: *Document Object Model*, es una interfaz para HTML, CSS y SVG que nos facilita una representación en forma de árbol sobre la que podremos trabajar con JavaScript.

BOM (Browser Object Model)

Es una recopilación de los componentes más utilizados, aunque existen algunos más.

window.history

Nos permite manipular el historial de la sesión actual, eso incluye solamente las páginas visitadas con la pestaña actual.

Podemos averiguar la cantidad de páginas visitadas previamente, haciendo uso de la propiedad *length*.

```
history.length
```

También podemos realizar acciones, como mandar al usuario a la pagina inmediatamente siguiente con el método *forward*, a la anterior con *back()* o directamente a cualquier posición del historial con *go()*.

```
// Ir atrás
history.go(-1);
history.back();

// Ir adelante
history.go(1);
history.forward();
```

window.navigator

Es un API que nos permite sacar gran cantidad de información sobre la máquina donde se está ejecutando nuestro script. Incluso dispone de algunos métodos tan interesantes como *Navigator.vibrate()*, que permite hacer vibrar el dispositivo (siempre que sea compatible).

En el siguiente ejemplo, hacemos una lectura de gran información del sistema y además hacemos un par de cálculos interesantes para confirmar el nivel de batería.

En la línea 28 hemos utilizado una *promesa*, ya que *navigator.getBattery() lo requiere así. Recuerda que ésta es otra de las formas válidas, que existen para manejar la asincronía.*

```
1  function conversorTiempo(segundos){
2          var fecha = new Date(segundos * 1000);
3          var hh = fecha.getUTCHours();
4          var mm = fecha.getUTCMinutes();
5          var ss = fecha.getSeconds();
6
7          if (hh < 10) {hh = "0"+hh;}
8          if (mm < 10) {mm = "0"+mm;}
9          if (ss < 10) {ss = "0"+ss;}
10
11         return hh+":"+mm+":"+ss;
12 }
13
14 function informacionSistema(){
15         console.log("appCodeName:", window.navigator.appCodeName);
16         console.log("appName:", window.navigator.appName);
```

```
        console.log("appVersion:", window.navigator.appVersion);
        console.log("platform:", window.navigator.platform);
        console.log("product:", window.navigator.product);
        console.log("userAgent:", window.navigator.userAgent);
        console.log("javaEnabled:", window.navigator.javaEnabled());
        console.log("language (used):", window.navigator.language);
        console.log("language (support):", window.navigator.languages);
        console.log("conectado a internet?", window.navigator.onLine);
        console.log("mimeTypes:",window.navigator.mimeTypes);
        console.log("Plugins:", navigator.plugins);

        navigator.getBattery().then(function(bateria){
                console.log("Batería cargando?", bateria.charging)

                if(bateria.charging){
                        console.log("Tiempo de carga:", bateria.chargingTime)
                }
                console.log("Batería %:", bateria.level*100)
                console.log("Tiempo restante:", conversorTiempo(bateria.dischargingTime))
        });

}
```

window.screen

Esta API nos permite sacar toda la información disponible de la pantalla (márgenes, profundidad del color…), así podremos bloquear y desbloquear la rotación de la pantalla en el dispositivo.

```
console.log("availTop:", window.screen.availTop);
console.log("availLeft:", window.screen.availLeft);
console.log("availHeight:", window.screen.availHeight);
console.log("availWidth:", window.screen.availWidth);
console.log("colorDepth:", window.screen.colorDepth);
console.log("height:", window.screen.height);
console.log("left:", window.screen.left);
console.log("orientation:", window.screen.orientation);
console.log("pixelDepth:", window.screen.pixelDepth);
console.log("top:", window.screen.top);
console.log("width:", window.screen.width);
```

Window.location y *Document.location*

Según la W3C ambos objetos son lo mismo.

Propiedades

```
var enlace = document.createElement('a');
enlace.href = 'https://leanpub.com/javascript-inspirate';

console.log("href:" ,enlace.href);
console.log("protocol:", enlace.protocol);
console.log("host:", enlace.host);
console.log("hostname:", enlace.hostname);
console.log("port:", enlace.port);
console.log("pathname:", enlace.pathname);
console.log("search:", enlace.search);
console.log("hash:", enlace.hash);
console.log("origin:", enlace.origin);
```

Métodos:

- ***.assign()***

  Carga una nueva página.

```
document.location.assign('https://leanpub.com/javascript-inspirate');
```

- ***.reload()***

  Recarga la página actual con opciones para manejar el cacheado.

```
// Recarga
document.location.reload();

// Recarga sin usar el cache
document.location.reload(true);
```

- ***.replace()***

  Carga una página nueva, sustituyendo la actual en el historial.

```
document.location.replace('https://leanpub.com/javascript-inspirate');
```

DOM

Veamos como entiende la comunidad *MDN (Mozilla Developer Network)* el concepto *DOM - Document Object Model*.



> El modelo de objeto de documento (DOM) es una interfaz de programación para los documentos HTML y XML. Facilita una representación estructurada del documento y define de qué manera los programas pueden acceder, con el fin de modificar, tanto su estructura, estilo y contenido. El DOM da una representación del documento como un grupo de nodos y objetos estructurados que tienen propiedades y métodos. Esencialmente, conecta las páginas web a scripts o lenguajes de programación.
>
> Una página web es un documento. Éste documento puede exhibirse en la ventana de un navegador o también como código fuente HTML. Pero, en los dos casos, es el mismo documento. El modelo de objeto de documento (DOM) proporciona otras formas de presentar, guarda y manipular este mismo documento. El DOM es una representación completamente orientada al objeto de la página web y puede ser modificado con un lenguaje de script como JavaScript.

Nosotros creemos que el DOM es patio de juegos de cualquier developer que se precie. Tanto si haces frontend y lo utilizas para mostrar información dinamicamente como si estás en el backend y lo utilizas para *scrapear*.

A los ojos de JavaScript el DOM es un objeto con el que podremos leer y modificar a nuestro antojo. Una vez se comprende con claridad la estructura de nodos que lo compone y sus métodos pricipales, está dominado.

Selectores

El primer paso para poder manipular el DOM, es adquirir cierta destreza en el manejo de los selectores, ya que siempre los selectores serán el primer paso, para realizar operaciones de lectura o modificación del DOM.

Es importante remarcar, que ciertos métodos para realizar selección de componentes, sufren del mismo problema que vimos en *arguments*, ya que aunque parecen *arrays*… realmente no lo son.

Estos métodos son:

- *Node.childNodes*
- *document.querySelectorAll*



Podemos convertirlos fácilmente:

```
var listaDivs = document.querySelectorAll('div');
// Conversión
var listaDivsArray = Array.prototype.slice.call(listaDivs);
```

Más información sobre la conversión en Convert NodeList to Array de David Walsh.

Selectores tradicionales

- ***.getElementById()***

  Permite la selección de un elemento por su *id*.

```
// <div id="miDiv"></div>
document.getElementById("miDiv");
```

- ***.getElementsByName()***

  Permite la selección de varios elementos por su atributo *name*.

```
// <form name="miForm"></form>
document.getElementsByName("miForm");
```

- ***.getElementsByTagName()***

  Permite la selección de varios elementos por su *etiqueta*.

```
// <input>
document.getElementsByTagName("input");
```

- ***.getElementsByClassName()***

  Permite la selección de varios elementos por su *clase*.

```
// <div class="rojo"></div>
document.getElementsByClassName("rojo");
```

Selectores Avanzados



Lecturas recomendadas:

- *Los 30 selectores CSS que debes memorizar*
- *Selectors Level 3 - W3C*
- *CSS Cheat Sheet*
- *Taming Advanced CSS Selectors by Inayaili de Leon*
- *Selectors en MDN*
- *CSS Selector Reference by w3schools*
- *CSS Selectors by Adam Roberts*
- *CSS-TRICKS Selectors*
- *Interactive CSS Selectors by Ben Howdle*

Si has trabajado intensamente con CSS3 ya sabrás que existen muchas posibilidades para realizar selecciones dentro de un documento html que van mucho más alla de la clase, id, etiqueta o propiedades, ya que *el soporte* para ello es muy bueno en todos los navegadores.

JavaScript no se queda atras y se pueden usar *querySelector* y *querySelectorAll* que además gozan de un gran *soporte*.

Veamos como se utilizan algunos selectores avanzados de CSS3:

- URL que empieza con "javascript:"

```
a[href^="javascript:"] {
        color:blue;
}
```

- URL que contiene "google.es"

```
a[href*="google.es"] {
        color:orange;
}
```

- URL que termina con ".pdf"

```
a[href$=".pdf"] {
        color:red;
}
```

.querySelector()

Devuelve el primer elemento que coincida con el selector.

```
<div id="miDiv">
    <span id="miId5" class="miClase" title="cinco"></span>
    <span id="miId4" class="miClase" title="cuatro"></span>
    <span id="miId3" class="miClase" title="tres"></span>
    <span id="miId2" class="miClase" title="dos"></span>
    <span id="miId1" class="miClase" title="uno"></span>
</div>
```

```
document.getElementById('miId1').title // uno
document.querySelector('#miDiv .miClase').title // cinco
document.querySelector('#miDiv #miId1.miClase').title // uno
document.querySelector('#miDiv .inventado').title // ERROR -> undefined
document.querySelector('#miDiv .miClase[title^=u]').title // uno
```

.querySelectorAll()

Devuelve todos los elementos que coincidan con el selector en un pseudo-array.

```
document.querySelectorAll('p') // los párrafos
document.querySelectorAll('div, img') // divs e imágenes
document.querySelectorAll('a > img') // imágenes contenidas en enlaces
```

Estilos con Javascript

JavaScript también puede leer y alterar las reglas de estilo, que afectan a los elementos que componen el DOM.

Leer valores

```
window.getComputedStyle(document.getElementById("id"));
window.getComputedStyle(document.body).getPropertyValue('display');
```

Cambiar valores

```
document.body.style.display="none";
document.getElementById("id").style.display="none";
```

Alterando el DOM

Trabajar sin JQuery

Si llevas mucho tiempo trabajando con JQuery, sin duda, ya estás muy acostumbrado a utilizar ciertos métodos como *show()*, *hide()*, *empty()*, *append()* y muchos más…

Para haceros la transición más fácil, *HubSpot* ha desarrollado *You Might Not Need Jquery*, que te permite visualizar como hacer todo lo que hacías con JQuery y que ahora harás con JavaScript.

Métodos esenciales

Hacemos una recopilación simplificada de los métodos más utilizados, aunque existen muchos más.

- ***.textContent***

  Nos devuelve el texto de un elemento previamente seleccionado.

```
var el = document.getElementById('miDiv');
console.log("Texto:", el.textContent);
```

  También permite cambiarlo.

```
var el = document.getElementById('miDiv');
el.textContent = "Nuevo contenido";
```

- ***.classList.contains()***

  Verifica si contiene cierta clase.

```
var el = document.getElementById('miDiv');
el.classList.contains("rojo");
```

- ***.classList.add()***

  Permite añadir una clase al elemento.

```
var el = document.getElementById('miDiv');
el.classList.add("rojo");
```

- ***.classList.remove(className)***

  Permite eliminar una clase del elemento.

```
var el = document.getElementById('miDiv');
el.classList.remove("rojo");
```

- ***.style.display***

  Nos permite conocer y modificar la visualización de un elemento concreto.

```
var el = document.getElementById('miDiv');

// Conocer el estado actual
console.log(el.style.display);

// No mostrar
el.style.display = 'none';

// Mostrar
el.style.display = '';
```

- ***.cloneNode()***

  Permite clonar un nodo y clonar o no, sus nodos hijo también.

```
var el = document.getElementById('miDiv');

// Clon simple (sin hijos)
var c1 = el.cloneNode();

// Clon profundo (con hijos)
var c2 = el.cloneNode(true);
```

- ***.innerHTML*** Cambia o devuelve la sintaxis HTML de un elemento… así como de sus hijos.

```
var el = document.getElementById('miDiv');
// Ver el contenido
console.log("Contenido:", el.innerHTML);

// vaciar el contenido
el.innerHTML = '';

// Sustituir el contenido
el.innerHTML = 'Nuevo contenido';
```

- ***.getAttribute()***

  Devolver el valor de cierto atributo o una cadena vacía en caso de no existir.

```
var el = document.getElementById('fotoUsuario');
el.getAttribute('src');
```

- ***.setAttribute()***

  Cambiar el valor de un atributo o lo añade si no existe.

```
var el = document.getElementById('miDiv');
el.setAttribute('data-secreto', 'Mucho...');
```

Eventos

Una de las mayores barreras a la hora de introducir el dinamismo en una web es la gestión de los eventos, ya que estos son asíncronos por naturaleza, por lo que no puedes saber de antemano. Por ejemplo, cuántos segundos tardará un usuario en pinchar sobre un botón… lo que sumado a un funcionamiento peculiar con propagaciones, hace que el dominio de los eventos sea una tarea complicada.



Sin embargo, todo gran esfuerzo tiene una gran recompensa y el dominio de estos eventos nos abrirá las puertas a la *programación dirigida a eventos*, y posteriormente a *Node.js*, si decidimos dar el gran paso hacia el *Back-End* o el *Full Stack*.

Funcionamiento

Básicamente podemos seleccionar un elemento de HTML, y suscribir uno de los posibles eventos de los que dispone:

- *Eventos de ratón*
- *Eventos de teclado*
- Y *muchos más…*

Una vez hemos definido estos detalles, en dónde escuchamos y qué esperamos, solo queda definir que haremos cuando esto ocurra.

Lógicamente al tratarse de asincronía, debemos definir una función, ya sea en línea o reutilizable para gestionarlo todo.

Si necesitamos información adicional sobre el evento sucedido, el propio sistema mandará *un objeto con detalles clave sobre el evento*, como argumento a la hora de ejecutar nuestro callback.

Utilizando eventos

Existen dos formas básicas de añadir eventos a nuestra aplicación. Una es por medio de html con atributos como *onclick*, y la otra desde el JavaScript, haciendo uso de métodos como *.addEventListener()*.

Cuál es mejor o cuál es peor… realmente varia en función de las circunstancias del código y su programador. Yo creo firmemente, que es mucho mejor separar JavaScript, HTML y CSS.

Así evitamos el *antipatrón* de mezclar JavaScript en el HTML. Por otra parte, si encapsulas tu código en una función anónima autoejecutada, no podrás acceder a las funciones directamente desde el HTML…

Así que por estos motivos principales, recomendamos encarecidamente el uso de *.addEventListener()*.

Añadir un evento

Veamos como funciona con *onclick*.

```
<body onclick="cambiarFondo()"></body>
```

```
function cambiarFondo() {
        // color = 'rgb(0-255,0-255,0-255)'
        var color = 'rgb(' + Math.floor((Math.random() * 255))+ ',';
        color += Math.floor((Math.random() * 255)) + ',';
        color += Math.floor((Math.random() * 255)) + ')';
        document.body.style.backgroundColor= color;
        console.info("Nuevo color:", color);
}
```

Veamos como funciona con *addEventListener()*.

```
document.body.addEventListener('click', function (e) {
    var color = 'rgb(' + Math.floor((Math.random() * 255))+ ',';
    color += Math.floor((Math.random() * 255)) + ',';
    color += Math.floor((Math.random() * 255)) + ')';
    document.body.style.backgroundColor= color;
    console.info("Nuevo color:", color);
});
```

Eliminar un evento

Los eventos ocupan memoria. Es un factor a tener en cuenta, especialmente, si queremos hacer una web con un buen soporte para smartphones. Si un evento deja de tener sentido, sencillamente puedes eliminarlo con *.removeEventListener()*.

```
function cambiarColor (){
    var color = 'rgb(' + Math.floor((Math.random() * 255))+ ',';
    color += Math.floor((Math.random() * 255)) + ',';
    color += Math.floor((Math.random() * 255)) + ')';
    document.body.style.backgroundColor= color;
    console.info("Nuevo color:", color);
}

document.body.addEventListener('click', cambiarColor);
document.body.removeEventListener('click', cambiarColor);
```

Manejadores de eventos

Cuando se dispara un evento, podemos recopilar mucha información útil, si no ignoramos el argumento que nos pasan.

Podemos fácilmente crearnos una función que nos aporte información real sobre el evento y que no ejecute ninguna acción en nuestra aplicación. Este tipo de funciones son muy útiles y deberían estar en tu kit de herramientas para desarrollar webs como un artesano de verdad.

```
function manejadorEventos(elEvento) {
          // Compatibilizar el evento
          var evento = elEvento || window.event;

          // Imprimir detalles
          console.log("-----------------------------")
          console.log("Type: "+evento.type); // Tipo
          console.log("Bubbles: "+evento.bubbles);
          console.log("Cancelable: "+evento.cancelable);
          console.log("CurrentTarget: ", evento.currentTarget);
        console.log("DefaultPrevented: "+evento.defaultPrevented);
        console.log("EventPhase: "+evento.eventPhase);
        console.log("Target: ", evento.target);
        console.log("TimeStamp: "+evento.timeStamp);
        console.log("IsTrusted: "+evento.isTrusted); // true - Usuario
        console.log("=============================")
}

// Añadimos Listener
document.addEventListener('click', function(evento){
        manejadorEventos(evento);
        // Más código
});
```

Usos Avanzados

Lanzar un evento manualmente con .dispatchEvent()

```
document.body.addEventListener('click', function (e) {
    var color = 'rgb(' + Math.floor((Math.random() * 255))+ ',';
    color += Math.floor((Math.random() * 255)) + ',';
    color += Math.floor((Math.random() * 255)) + ')';
    document.body.style.backgroundColor= color;
    console.info("Nuevo color:", color);
});

var lanzadorEventos = new Event('click');
document.body.dispatchEvent(lanzadorEventos);
```

Compatibilidad con Internet Explorer <= IE8

Si tienes la mala suerte de tener que dar soporte a versiones obsoletas de Internet Explorer, además de todo el sufrimiento acumulado, deberás añadirte una carga extra… y es que por aquel entonces Internet Explorer, no era compatible con *addEventListener()*, ni con *.removeEventListener()*, así que necesitamos utilizar *.attachEvent()* y *.detachEvent()*.

.attachEvent()

```
document.attachEvent('onclick', function (e) {
    var color = 'rgb(' + Math.floor((Math.random() * 255))+ ',';
    color += Math.floor((Math.random() * 255)) + ',';
    color += Math.floor((Math.random() * 255)) + ')';
    document.body.style.backgroundColor= color;
    console.info("Nuevo color:", color);
});
```

.detachEvent()

```
function cambiarColor (){
    var color = 'rgb(' + Math.floor((Math.random() * 255))+ ',';
    color += Math.floor((Math.random() * 255)) + ',';
    color += Math.floor((Math.random() * 255)) + ')';
    document.body.style.backgroundColor= color;
    console.info("Nuevo color:", color);
}

document.body.attachEvent('onclick', cambiarColor);
document.body.detachEvent('onclick', cambiarColor);
```

Dado este panorama de tener que usar dos métodos diferentes, cuando además tienen una estructura similar, pero no idéntica… tenemos un reto entre manos que merece una solución duradera.

Aunque descartamos enseñar el uso de *patrones*, sería injusto por nuestra parte no arrojar al menos un poco de luz sobre el asunto, así que nos gustaría mostrarte un par de trucos de *“artesano veterano”*.

Vamos a crear un objeto (literal), donde guardaremos todo lo necesario para manejar eventos, subiendo así el nivel de abstracción de vuestra aplicación.

Ya no usaremos *.attachEvent()* o *.addEventListener()*. Ahora utilizaremos *.agregar()*, que internamente llamará a *.attachEvent()* o *.addEventListener()*, en función del navegador en el que nos encontremos.

Cambiar de navegador en medio de la ejecución del script parece altamente improbable. No tiene mucho sentido comprobar cada vez que agreguemos un evento, si usaremos *.attachEvent()* o *.addEventListener()*.

Por eso los patrones de diseño pueden ayudarnos a solucionar cosas así. En este caso usaremos *Init-time branching*, que es una variante de *Lazy initialization*.

Básicamente crearemos un objeto con los métodos *agregar* y *quitar* vacíos. Al ejecutarse el código por primera vez, comprobaremos que navegador usamos y en función de ello sobreescribirá *agregar* y *quitar*, llamando por debajo a los métodos correspondientes.

Es un poco más de trabajo de lo esperado, pero así es mucho mas sencillo gestionar nuestra aplicación, ya que nuestros métodos se adaptarán y no así nuestro código.

El valor añadido de usar este patrón está en que en un solo punto tomo una decisión que se extenderá por toda nuestra aplicación.

```
var eventos = {
        agregar: null,
        quitar: null,
        manejador: function(evento) {
        console.group("Manejador de Eventos");
        console.log("------------------------------");
        console.log("Type: " + evento.type); // Tipo
        console.log("Bubbles: " + evento.bubbles); // sube por el DOM
        console.log("Cancelable: " + evento.cancelable);
        console.log("CurrentTarget: ", evento.currentTarget);
        console.log("DefaultPrevented: " + evento.defaultPrevented);
        console.log("EventPhase: " + evento.eventPhase);
        console.log("Target: ", evento.target);
        console.log("TimeStamp: " + evento.timeStamp);
        console.log("IsTrusted: " + evento.isTrusted); // true - Usuario
        console.log("==============================");
                console.groupEnd();
        }
```

```
}

// Init-time branching (Patrón)
if (typeof window.addEventListener === 'function') {
    eventos.agregar = function(el, type, fn) {
            el.addEventListener(type, fn, false);
        };
        eventos.quitar = function(el, type, fn) {
                el.removeEventListener(type, fn, false);
        };
} else { // Soporte para <= IE8
    eventos.agregar = function(el, type, fn) {
        el.attachEvent('on' + type, fn);
    };
    eventos.quitar = function(el, type, fn) {
        el.detachEvent('on' + type, fn);
    };
}

eventos.agregar(document.body, 'click', function (e) {
        var color = 'rgb(' + Math.floor((Math.random() * 255))+ ',';
        color += Math.floor((Math.random() * 255)) + ',';
        color += Math.floor((Math.random() * 255)) + ')';
        document.body.style.backgroundColor= color;
        console.info("Nuevo color:", color);
})
```

Delegación de Eventos

Una de las técnicas más útiles para ahorrar memoria, es utilizar la *delegación de eventos*, es decir, en vez de poner un evento por cada uno de los elementos que compone una estructura de datos, es preferible hacerlo únicamente, sobre el elemento padre común a todos ellos y simplemente filtrar cuál de los hijos fue disparado.

Modo Clásico (sin delegación)

```
<ul id="miNav">
        <li><a href="#nosotros">¿Quienes Somos?</a></li>
        <li><a href="#objetivos">Los objetivos</a></li>
        <li><a href="#equipo">Nuestro Equipo</a></li>
        <li><a href="#detalles">Más detalles</a></li>
        <li><a href="#contacta">Contactanos</a></li>
</ul>
```

```
var miNav = document.getElementById("miNav");
var miNavLinks = miNav.getElementsByTagName("a");

for (var i = 0; i < miNavLinks.length; i++) {
    miNavLinks[i].onclick = function(){
            console.info(this.innerHTML);
        }
};
```

Modo Delegando

```
var miNav = document.getElementById("miNav");
miNav.onclick = function(evento){

var evento = evento || window.event;
var elemento = evento.target || evento.srcElement;
    console.info(elemento.innerHTML);
};
```

Creación de Eventos Personalizados

Otra estrategia más avanzada, es hacer nuestro código más modular e interconectado por eventos, aunque esto es más típico de Node.js. En el lado del cliente también podemos hacer uso de ello.



Como verás incluso podemos disparar eventos, lo que amplia mucho las posibilidades para jugar con páginas web existentes desde la consola… ***¡Yo no digo nada…!***

```
var evento = new Event('miEventoInventado');

document.body.addEventListener('miEventoInventado', function (e) {
        console.info(e); // {isTrusted: false}
});

document.body.dispatchEvent(evento);
```

Propagación (Capturing y Bubbling)

Uno de los puntos de fricción más altos a la hora de trabajar con eventos, es entender como funciona la propagación que JavaScript realiza.

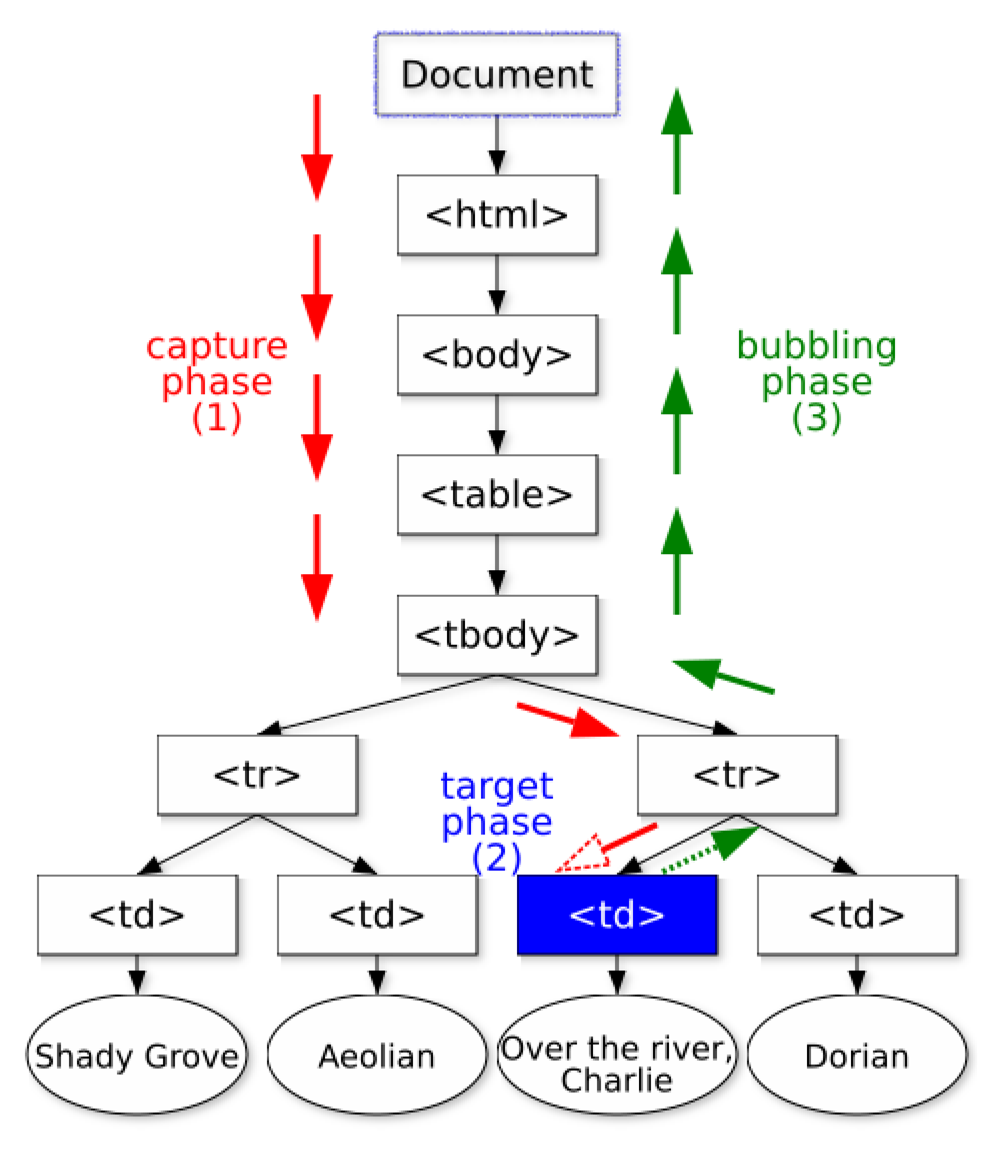


Imagen obtenida de W3C

La mejor manera de entenderlo es con un *ejemplo*.

Si ejecutas el evento verás que existe un pequeño conflicto de intereses en el HTML.

```
<div id="parent">
    <div id="children">
        Click
    </div>
</div>
```

Tanto *parent* como *children* tienen eventos suscritos. El problema básico, que nos encontramos es que *children* está envuelto por *parent*, lo que hace que pinchemos, donde pinchemos siempre se disparan ambos.

Esto se debe a la propagación, que funciona en tres fases:

Capturing
: Recorre todo el documento desde *document* hasta el elemento en sí.

Target
: Es cuando se llega al elementos en cuestión.

Bubbling
: Realiza el recorrido de vuelta hacía arriba desde el elemento en sí.

Podemos interrumpir este proceso de propagación si hacemos uso de *stopPropagation()*.

Como has podido ver, el ejemplo en sí que hemos utilizado aquí, fuerza un poco la situación, ya que existe mucho código duplicado y gestiona los eventos de una manera incorrecta.

Otro método interesante, que no debemos perder de vista es *.preventDefault*. Evita el comportamiento por defecto, por ejemplo si queremos que un link no actúe como tal.

Al contrario de lo que pueda parecer ahora mismo, la propagación es uno de los mejores aliados a la hora de gestionar nuestros eventos, como vimos anteriormente con la *delegación de eventos*.

Capítulo 12 - AJAX y más AJAX

Entendiendo HTTP/s

Http *(Hypertext Transfer Protocol)* es un protocolo de comunicación que utilizamos en nuestro día a día. Una de sus características clave es que no *mantiene el estado*, que ocasiona que tengamos que buscar alternativas para ello como las *cookies*, *localStorage*, etc…

Como curiosidad la versión más utilizada de este protocolo es la 1.1 de 1999, aunque recientemente, tenemos ya liberada la versión 2 que incluye muchas mejoras en el empaquetamiento y transporte de la información.

La parte clave a la hora de entender este protocolo, es que sigue un esquema petición-respuesta entre cliente y servidor, que hace que tengamos que refrescar la página para poder recibir la nueva información. Tal como funcionaba la web a principios del dos mil…

Si nos adentramos un poco más, veremos que esa peticion-respuesta, básicamente se compone de mensajes con una estructura muy determinada.

Línea inicial
: En la petición se especifica el método utilizado.
: En la respuesta se especifica un código de respuesta.

Cabecera
: Podremos encontrar todos los metadatos propios de la petición y la respuesta.

Cuerpo
: Esta parte es opcional y contiene por ejemplo html, css… textos planos, lo que sea en si el mensaje desde el servidor, así como desde el cliente.

Métodos

Cuando navegamos por la web utilizando el navegador por defecto, siempre estamos haciendo uso del método *GET*, pero existen gran cantidad de métodos con lo que poder hacer las peticiones.

Esto es importante, ya que desde las peticiones AJAX podremos especificar el método que utilizaremos, con lo que si estamos trabajando con una API REST, veremos que existe cierta “correlación” entre las rutas y los métodos usados.

Los métodos más utilizados son *GET*, *POST (típico de formularios)*, *PUT* y *DELETE*.

Códigos de error

Cuando nos llega la respuesta del servidor, siempre viene acompañada de un código numérico que nos confirma el resultado de la petición, por ejemplo, el *mítico error 404*, que nos habremos encontrado en multitud de ocasiones navegando por la red.

Estos códigos de estado pueden dividirse en cinco categorías principales:

- 1xx **Informativas**.
- 2xx **Peticiones Correctas**.
- 3xx **Redirecciones**.
- 4xx **Errores Cliente**.
- 5xx **Errores Servidor**.

Siendo especialmente importantes para nosotros, los 4xx y los 5xx, ya que nos indican errores que están sucediendo y esto seguramente derive en un tráfico de datos erróneo.



Aquí podéis encontrar la *lista completa* y la *especificación*.



Misterioso estado 418

Existen algunos códigos tan increíbles como *418. I´m a teapot*.

Trabajando con APIs

Es innegable que cada día es más popular, escuchar términos como *API*, *API REST*… esto se debe a la manera de hacer la web hoy en día, con un gran desacoplamiento de cliente y servidor, que permite a un mismo backend, dar soporte a multiples soluciones de front, ya sean web, móviles, *Internet Of Things (IoT)*… e incluso otros backends.

Para hacer que todo esto fluya, es necesario comprender que juegan muchos factores claves como:

- ***API***
- ***REST***
- ***SOAP***

Normalmente en el mundo de la web cuando nos referimos a una API, casi siempre hablamos de una API REST, que nos permite a través de una serie de urls y siguiendo una *metodología CRUD*, trabajar con un backend desde el cliente.

Esto permite conseguir datos (peticiones GET), actualizar datos (peticiones PUT), crear nuevos datos (peticiones POST) y eliminar datos (peticiones Delete).

En la mayoría de los casos, el intercambio de información entre el cliente y servidor se realiza utilizando un formato de intercambio de datos conocido como JSON y no XML como antiguamente. De todo esto hablaremos a lo largo de este último capítulo.



Para ilustrar un poco el ejemplo, te invito a visitar *jsonplaceholder* que es una especie de *Lorem Ipsum* para peticiones Ajax.

Supongamos que nuestro backend es un blog, y queremos crear, leer, actualizar y borrar posts. Nuestras rutas serían algo así:

Petición GET a *api.loremblog.com/posts*
: Nos devuelve un JSON con todos los post disponibles.

Petición POST a *api.loremblog.com/posts*
: Nos crea una nueva entrada en la base de datos, con los datos en JSON que le hemos pasado

Petición PUT a *api.loremblog.com/posts/hello*
: Nos permite actualizar la entrada (numero/nombre hello) con los datos JSON que hemos pasado.

Petición DELETE a *api.loremblog.com/posts/hello*
: Nos permite borrar la entrada (numero/nombre hello).



Partiendo de esta filosofía, podremos atrevernos con muchas APIs, tan divertidas como Twitter, *Facebook*, *Spotify* e incluso *Yahoo!*

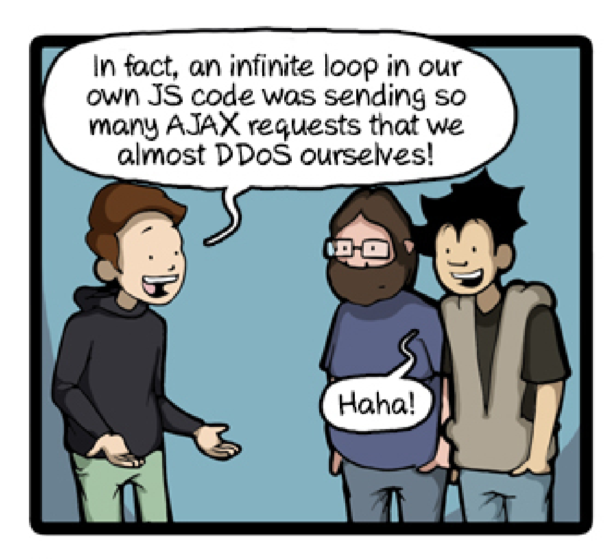
Peticiones AJAX

Ya hemos visto que las peticiones Ajax son la esencia en la web de hoy, pero siempre que las utilicemos, debemos recordar, que nosotros somos consumidores/generadores de datos y que nuestra aplicación, está dependiendo de manera permanente de otros sistemas.

Esto quiere decir, que puede haber errores en su código, que hagan que las respuestas a nuestras peticiones no traigan los datos como esperamos. Por tanto, tendremos que hacer un esfuerzo adicional para validar todos los datos que recibimos. En ocasiones la documentación sobre la API, puede ser poco precisa o estar desfasada… dejándonos en una situación muy comprometida.

Pero la mayoría de servicios grandes o empresas que orientan sus líneas de negocio al API suelen dar un buen servicio. Recordar también que existen APIs de pago. En ocasiones esto puede resultar complicado, si estamos desarrollando, tenemos que hacer muchas consultas y se dispone de poco presupuesto.

Nosotros recomendamos, crear un sistema de archivos JSON que almacene ciertos datos estáticos, que sean iguales o muy similares a los que esperamos recibir de la API. Así podremos maquetar y desarrollar sin consumir realmente llamadas. Esto es especialmente útil, para sistemas muy caros como APIs de *Inteligencia Artificial como Servicio (AIaaS)*.





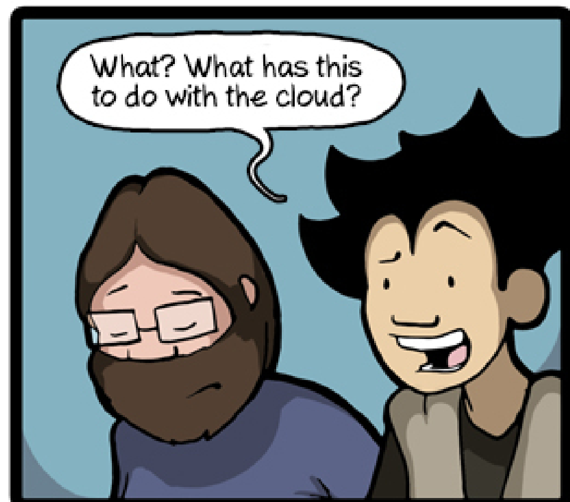




Imagen de CommitStrip.com

Errores más comunes (por código)

Referimos a continuación una lista de los errores más comunes, y de algunas soluciones cuando no encontramos mucha documentación disponible sobre una API en concreto.

Respuesta 200
: *OK*. Este es el código esperado… y todo marcha bien.

Respuesta 204
: *Sin contenido*. No hemos enviado los datos, pero la petición ha resultado exitosa, seguramente con datos por defecto.

Respuesta 401
: *No autorizado*. Hemos olvidado utilizar el token. O nuestro token no está correctamente configurado.

Respuesta 403
: *Prohibido*. No tenemos el token/acceso correcto. Puede ser que intentemos modificar elementos que no sean de nuestra competencia, como la configuración de otros usuarios, etc…

Respuesta 404
: *No encontrado*. Hemos apuntado a una ruta que no existe. Revisa la documentación.

Respuesta 409
: *Conflicto, ya existe*. Hemos intentado crear dos veces el mismo elemento, esto suele pasar si realizamos una doble llamada Ajax, por error. Típico de peticiones *POST*.

JSON

Antes de realizar nuestra primera llamada Ajax, veamos como trabajar con JSON.

Para la visualización, validación y conversión dinámica de datos contamos con algunas herramientas muy útiles.

- *JSONLint*
- *JSON Formatter & Validator*
- *Online JSON Viewer*
- *JSON Editor*



Es importante recordar que algunas respuesta JSON, pueden contener hasta varios megas de información. Necesitaras herramientas de visualización adicionales más allá de la consola de tu navegador.

El navegador nos aporta dos formas básicas de interactuar con datos JSON.

- ***JSON.parse()***

  Que analiza la cadena JSON, validándola y retornando los valores como un dato válido de JavaScript. Lo más habitual es un objeto o array.

```
var objeto = JSON.parse('{"test": true}');
var lista = JSON.parse('[true, "NO", 23]');
var booleano = JSON.parse('false');
var texto = JSON.parse('"cadena!"');
```

- ***JSON.stringify()***

  Analiza los valores y retorna una cadena en formato JSON.

```
var objetoJSON = JSON.stringify({"test": true});
var listaJSON = JSON.stringify([true, "NO", 23]);
var booleanoJSON = JSON.stringify(false);
var textoJSON = JSON.stringify("cadena!");
```

Peticiones

La mecánica de las peticiones es realmente sencilla. Simplemente necesitamos saber a que URL vamos a realizar la petición, que método vamos a utilizar (GET, POST, PUT o DELETE) y con que función (callback) manejaremos los datos de vuelta.



Lógicamente las peticiones Ajax son asíncronas por naturaleza. Si aún tienes dudas, sobre los callbacks y el manejo de la sincronía, este es un gran momento para retroceder al capítulo anterior y repasar esos conceptos antes de seguir.

Si estáis familiarizados con JQuery, las peticiones Ajax son algo muy trivial, gracias a *jQuery.ajax()*. Veamos el siguiente ejemplo:

```
$.ajax({
    dataType: "json",
    url: "<---URL---->",
})
 .done(function( data, textStatus, jqXHR ) {
     console.log( "La solicitud se ha completado correctamente." );
     console.log( data );
 })
 .fail(function( jqXHR, textStatus, errorThrown ) {
      console.log( "La solicitud a fallado: " +  textStatus);
});
```

No recargamos pero somos asíncronos.

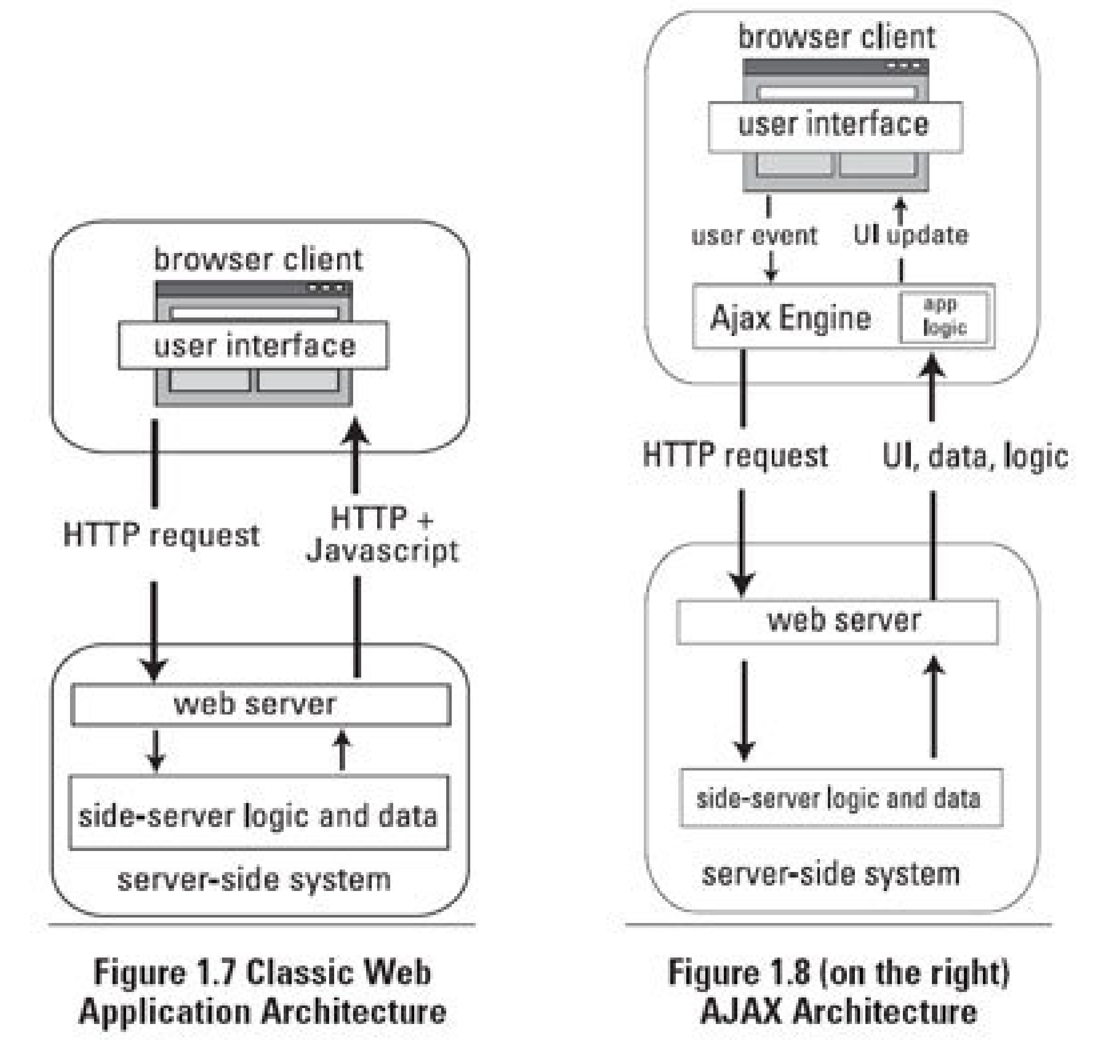


Imagen de gemsres.com

Como puedes ver, ya no es necesario recargar la página como antes, ya que el navegador se encarga de hacer la petición y gestionarla en un segundo plano. Ahora veremos como se hace con *Vanilla.js*, verás que realmente se entiende mejor el proceso y parece menos *mágico* que con JQuery.

Hablemos de *XMLHttpRequest*

Para poder realizar peticiones Ajax, de manera nativa, los navegadores implementan un objeto desde el que podremos realizar las peticiones, llamado *XMLHttpRequest*.

Tiene sus propios eventos:

- *loadstart*
- *progress*
- *abort*

- *error*
- *load*
- *timeout*
- *loadend*
- *readystatechange*

Y sus propios estados de petición:

- 0 *uninitialized*
- 1 *loading*
- 2 *loaded*
- 3 *interactive*
- 4 *complete*

Mejor veamos como funciona con un ejemplo real y luego explicamos como gestionar algo que parece tan complejo, pero que realmente es sencillo.

Para este ejemplo hemos decidido traer los datos de la *película Hackers*, que además os *recomiendo ver*.

Para ello utilizaremos *OMBd Api*… una vez leída la documentación, sabremos que solo necesitamos apuntar a esta url, *http://www.omdbapi.com/?t=Hackers&y=&plot=short&r=json*.

```
1  var url = "http://www.omdbapi.com/?t=Hackers&y=&plot=short&r=json";
2  var xmlHttp = new XMLHttpRequest();
3
4  xmlHttp.onreadystatechange = function() {
5
6      if (xmlHttp.readyState === 4)
7         if(xmlHttp.status === 200) {
8            console.info(JSON.parse(xmlHttp.responseText));
9         } else if (xmlHttp.status === 404) {
10           console.error("ERROR! 404");
11           console.info(JSON.parse(xmlHttp.responseText));
12     }
13 };
14 xmlHttp.open("GET", url, true);
15 xmlHttp.send();
```

Viendo el código es mucho más sencillo…

Vamos por pasos:

1. *Línea 1:* Primero definimos la URL.
2. *Linea 2:* Luego instanciamos *XMLHttpRequest*, igual que hacíamos con *Date* en capítulos anteriores.

3. *Linea 4:* Cada vez que cambiemos de estado en nuestra petición Ajax, el navegador ejecutará *xmlHttp.onreadystatechange* .
    1. *Linea 6:* Filtramos por el *xmlHttp.readyState*, y esperamos al 4 (completado).. obviando todo lo demás.
    2. *Linea 7:* Una vez confirmamos que se terminó toda la gestión de Ajax, pasamos a analizar los datos recibidos. Filtrando por código de estado con *xmlHttp.status*, y parseando o alertando al usuario del error en función de las circunstancias.
4. *Linea 14:* Abrimos la petición con la url y el método deseado.
5. *Linea 15:* Ejecutamos la petición.

CORS

Un problema clásico a tener en cuenta es la posibilidad de que *CORS*, no esté habilitado en el servidor.

CORS básicamente, es un parámetro que se envía desde las cabeceras HTTP del servidor y nos permite realizar peticiones AJAX. Existen algunos sitios web, que a pesar de tener, una API funcionando no han habilitado esta configuración:

```
"Access-Control-Allow-Origin", "*"
"Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept"
```

Y por ello, cualquier petición que hagamos desde el navegador cliente no podrá realizarse.

La mejor solución, es comunicarnos con los responsables de la API y facilitarles *este link*, donde *Monsur Hossain* y *Michael Hausenblas*, explican como realizar las configuraciones adecuadas en muchos entornos.

Si por desgracia esta opción está descartada, no pasa nada…. existen más posibilidades.

La primera y más útil es crear/configurar/usar un *proxy*. La idea del proxy, es que realice una petición capturando esos datos y enviándolos de nuevo, teniendo CORS habilitado en la cabecera, de tal forma que podremos hacer peticiones Ajax, aunque tengamos que pasar por nuestro servidor.

Lógicamente nos plantea un problema de tráfico, en nuestro servidor y una lentitud en nuestras peticiones, pero al menos podremos hacer las peticiones.



Antes de configurar un proxy, puedes usar *Crossorigin* para prototipar -es un *proyecto Open Source*- y podrás adaptarlo fácilmente a tus necesidades.

Parte IV - Un pasito más...

Anexo: ¡Queda mucho más por aprender!

Recursos

A lo largo del libro hemos incluido gran cantidad de links para facilitarte recursos para aprender más acerca de conceptos claves.

Hemos recopilado todos los enlaces de manera automática en esta lista de enlaces para vosotros en un solo lugar, *extras/recursos.md* en *nuestro repositorio*.

Libros interesantes



Deberías leer…

- *JavaScript for PHP Developers: A Concise Guide to Mastering JavaScript by Stoyan Stefanov*
- *You Don't Know JS (book series) by Kyle Simpson*
- *JavaScript Allongé, the "Six" Edition by Reg "raganwald" Braithwaite*
- *Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript by David Herman*
- *JavaScript for Kids: A Playful Introduction to Programming by Nick Morgan*
- *Eloquent JavaScript de Marijn Haverbeke*
- *JavaScript: The Good Parts de Douglas Crockford*
- *JavaScript Patterns by Stoyan Stefanov*
- *JavaScript Ninja de John Resig y Bear Bibeault*
- *JavaScript. La Guía Definitiva de David Flanagan*
- *Código Limpio de Robert C. Martin*
- *Sams Teach Yourself Node.js in 24 Hours by George Ornbo*
- *Learning JavaScript Design Pattern by Addy Osmani*

Ampliar horizontes

Esperamos que este libro os haya resultado de utilidad y que os haya ayudado a empezar a recorrer el largo camino del desarrollador web.

Os proponemos algunas ideas para que podáis seguir creciendo y evolucionando, a pesar de que nuestra historia acaba aquí.

Está claro que la mejor manera de aprender un lenguaje de programación es practicando, practicando y seguir practicando…

Éstas son algunas formas sencillas pero efectivas de lograrlo:

Aprender a utilizar Git y GitHub

Si queréis formar parte de la comunidad o entrar a trabajar en un entorno ágil, uno de los primeros pasos es aprender a manejaros con el control de versiones.

Aprender un nuevo lenguaje

Puede parecer una locura empezar con otro lenguaje, pero cada lenguaje tiene algo único y un enfoque especial. Aprender más lenguajes os ayudará a pensar mucho más allá del lenguaje que uses. Lo que a la larga hará que seáis un mejores artesanos.

Dominar un framework

Es una decisión complicada… un framework es un recurso, que puede ser difícil de aprender y en ocasiones tiene una vida útil muy corta. Gran parte del trabajo será elegir el framework que más os convenga. Es muy importante, tener en cuenta, que si es el primer framework… os enseñará una nueva forma de ver las cosas y de trabajar en JavaScript. No lo desaprovechéis, merece la pena emplear un poco más de tiempo en aprender nuevas formas de hacer las cosas.

Familiarízate con las librerías más populares

JQuery puede ser un gran comienzo. Podéis adaptar librerías a JQuery o al revés, así rápidamente ganaréis soltura. Existen muchas otras librerías que deberíais conocer y que te harán la vida más fácil.

Contribuye a mejorar las cosas

Puedes unirte a grupos de trabajo y organizaciones en Github e ir colaborando en alguno de los muchos proyectos que existen sobre JavaScript.

Documentar no es una mala idea

Durante la lectura de este libro, espero que hayáis podido consultar la *documentación de MDN* para poder aprender más sobre lo que os contábamos…. Recuerdar que esa documentación la crean muchos voluntarios, en todo el mundo, que entre otras cosas, la mejoran, amplían y traducen a diversos idiomas. No lo dudéis… **¡uniros!**

Ayuda al prójimo

Seguro que has mirado -más de una vez- como se hace algo en *Stackoverflow*, *Codepen*, foros especializados, comunidades, etc… Podéis también contribuir y ayudar a los demás, solucionando dudas. Al principio se hace muy cuesta arriba, pero luego es extraordinariamente atractivo. Y si no que le pregunten a *Coma*.

Guías de estilo

Adoptar una *guía de estilos* ayudará no solo a consolidar una forma correcta y uniforme de desarrollar código en JavaScript si no que también os ayudará a gestionaros con *La Ley de Parkinson de la Trivialidad*.

Recomendamos usar *Idiomatic.js* aunque existen muchas más como *JavaScript Style Guide de Airbnb* y *muchas más*.

Anexo: Comunidad...

¡Forma parte!

Meetups

Una de las maneras más efectivas de entrar en contacto con la comunidad y formar parte de un grupo de personas con el mismo interés, es *Meetup*.

Existen muchos grupos dedicados al desarrollo. Estos son solo algunos de los más activos.



Deberías conocer...

- *Open Source Weekends*
- *MadridJS*
- *Nodejs*
- *WordPress Madrid*
- *Hackathon Todos Incluidos*
- *Girls in Tech Spain*
- *Hackathon Lovers*
- *HTML5 Spain*
- *API Addicts*
- *ReactMad*
- *Edupreneurs Madrid*
- *IoT Madrid*
- *Angular Madrid*
- *GDG Madrid*
- *APP A Month*
- *Docker Madrid*
- *Madrid Python*
- *Front-End Developers Madrid*
- *Software Craftsmanship Madrid*
- *Tetuan Valley*
- *JS Dojo Madrid*
- *Polymer Madrid*
- *ReactJS Spain*
- *AdaLab*
- *FrontGirls*

Otras formas

También hay una gran oferta de eventos fuera de Meetup, como son:



Deberías conocer…

- *Campus Madrid Events*
- *Betabeers*
- *MediaLab Prado*

Desarrolladores que deberías seguir.

Esta lista se regenera de manera automática -cada vez- que se actualiza el libro. Recoge la información pública directamente de Twitter y la organiza por orden alfabético, tomando como referencia el *@userId*.



La lista no pretende ser exhaustiva ni completa en absoluto, por supuesto, no está todo el mundo y es meramente indicativa.

- ***Ariya Hidayat***

  @AriyaHidayat: *Software Provocateur. Author of PhantomJS and Esprima. Running engineering at @ShapeSecurity.*
- ***BrendanEich***

  @BrendanEich: *Brendan Eich invented JavaScript, co-founded http://mozilla.org , and has founded a new startup, https://brave.com/ .*
- ***Simeon.__proto__***

  @DotProto: *Oh, hello there. I'm Simeon , a Front End Engineer at @Blizzard_ent. JS lover, NIH fighter, and smitten with the open web. Opinions are my own.*
- ***John Papa***

  @John_Papa: *Husband, father, and Catholic enjoying every minute with my family. Disney fanatic, evangelist, HTML/CSS/JavaScript dev, speaker, and Pluralsight author.*
- ***Matias Arriola***

  @MatiasArriola: *When you see me again, it won't be me.*
- ***Pascal Precht ▯•▯▯•▯***

  @PascalPrecht: *I like headphones, art, skateboarding and coding. Angular GDE at @Google, @thoughtram co-founder and creator of @5thingsAngular*
- ***Sacha Greif***

  @SachaGreif: *Originally from Paris, but now living in Osaka. I co-wrote @DiscoverMeteor, and created @Telescpe and @SidebarIO. Addicted to #Hearthstone and #BJJ.*
- ***Simo Ahava***

  @SimoAhava: *Senior Data Advocate at @ReaktorNow | Google Analytics | Google Tag Manager | Passionate blogger | Experienced speaker | Google Developer Expert*

- ***Addy Osmani***

  @addyosmani: *Engineer at Google working on @GoogleChrome • Author • Creator of TodoMVC, @Yeoman, Material Design Lite, Critical • Husband*
- ***Amjad Masad***

  @amasad: *Founder @replit. Previously JavaScript @facebook (@babeljs, @reactnative)*
- ***xno⊠⊠⊠ u⊠ı⊠q***

  @brianleroux: *starting something new @begin *
- ***Chris Heilmann***

  @codepo8: *Londoner, German, European. Developer Evangelist - all things open web, writing and helping. Works at Microsoft on Edge, opinions totally my own. #nofilter*
- ***cody lindley***

  @codylindley: *Senior UI Engineer at Citrix 4 @Grasshopper. In all I do lover of Christ, people, logic & dying art of debate, conversation, & reason. Husband, father to 3 boys*
- ***Carlos Hernández***

  @codingcarlos: *Me levanto y programo. Bueno, programo sentado. Gamifico cosas, a veces prestidigito y hago música rara. Papi de @gamify_es*
- ***David Walsh***

  @davidwalshblog: *Mozilla Sr. Web Developer, Front-End Engineer, MooTools Core Developer, Javascript Fanatic, CSS Tinkerer, Node Hacker, web, VR, and open source lover.*
- ***Dejan Dimic***

  @dejan_dimic: *Minds are like parachutes - they only function when open. - Thomas Dewar*
- ***Dan Shaw***

  @dshaw: *CTO and Co-Founder of @NodeSource - secure, reliable, extensible Node.js. Always bet on #nodejs!*
- ***Duc Nguyen***

  @ducntq: *Khổ vì công nghệ*
- ***Elijah Manor***

  @elijahmanor: *{priorities:['Christian','Family','Work'], work:['@LeanKit', '@PluralSight', '@egg-headio'], tech:['#JavaScript', '#ReactJS','#jQuery'], titles:['@MVPAward']}*
- ***Erik Aybar***

  @erikthedev_: *Husband/Father • Remote Worker • Software Engineer @PracticeGenius • @eggheadio • Javascript, #reactjs, occasional ranting…*
- ***Firede***

  @firede: *…*
- ***Miloš Gavrilović***

  @gavrisimo: *Freelance web developer or something like that…*
- ***Richard Gibson***

  @gibson042: *Software engineer extraordinaire (@Dyn Architect, @jquery Core/Sizzle/-QUnit), polymath, übergeek, and passionate agorist.*
- ***Aleksandr Filatov***

  @greybax: *#frontend web developer*

- ***Idan Gazit***

  @idangazit: *Designer/Developer hybrid. @djangoproject core alum. Datavis junkie. Maker of things delightful. I like to ogle details. Extracting meaning from data @Heroku.*
- ***james young***

  @jamsyoung: *Applications Architect at Turner Broadcasting and developer on a variety of open source projects*
- ***John-David Dalton***

  @jdalton: *JavaScript tinkerer, bug fixer, & benchmark runner • Creator of Lodash • Former Chakra Perf PM • Current Web Apps & Frameworks PM @Microsoft.*
- ***John Resig***

  @jeresig: *Creator of @jquery, JavaScript programmer, author, Japanese woodblock nerd (http://ukiyo-e.org), work at @khanacademy.*
- ***jfroffice***

  @jfroffice: *Web Design, HTML5, ES6, SASS, CSS3, NodeJS, Angular, React, D3, JQuery, Docker, MongoDB, Nginx, Apache, Varnish*
- ***Kahlil Lechelt***

  @kahliltweets: *JavaScript person. Co-host of @reactivepod. Deejay. kahlilsnaps on Snapchat & kahlillechelt on IG. Frontend Architect at 1&1. Opinions are mine.*
- ***Kent C. Dodds***

  @kentcdodds: *Making software dev more accessible · Mormon, Husband, Father, Teacher · OSS, GDE, @TC39 · @PayPalEng @eggheadio @FrontendMasters @JavaScriptAir @React30 · #JS*
- ***Ulises Gascón***

  @kom_256: *#Maker, #ComputerGeek, #WebDeveloper, #IoT | #Js, #Node, #Python | Co-organizador de @os_weekends | Profe en @fictiziaescuela | xIBMer*
- ***Hao-Wei Jeng***

  @l0ckys: *lockys on the internet, loving in #WebDevelop #FrontEnd #JavaScript #NodeJS #brainwarer*
- ***Tracy Lee | ladyleet***

  @ladyleet: *Speaker|#GoogleDevExpert|Cofounder @thisdotmedia|Sold last startup|@ModernWeb_- @ngcruise @rxworkshop @venturehacked @embermeetup #foodie http://ladyleet.com*
- ***Leonardo A. Souza***

  @leobetosouza: *Chato. Carioca suburbano morando em Niterói. Cristão. Flamenguista. Web Developer. Chato.*
- ***Marco Trulla***

  @marcotrulla: *#Freelance #WebDev #Graphics #NodeJS #Photo #WebDeveloperItaliani (http://bit.ly/1jz9IG7) - #WebDesignerItaliani (http://bit.ly/1o4yNe7)*
- ***marocchino***

  @marocchino: *ruby, javascript, vim, elixir, elm*
- ***Mathias Bynens***

  @mathias: *Web standards fanatic. JavaScript, HTML, CSS, HTTP, performance, security, Bash, Unicode, macOS.*

- ***Mihai Paun***

  @mihaipaun: *Web craftsman. Art aficionado. Music head. Husband of @Be_InSight / http://in-sight.ie/*
- ***NatalieMac***

  @nataliemac: *I'm a web developer, UI designer, and founder of the amazing creative agency Purple Pen Productions.*
- ***Nick Salloum***

  @nicksalloum_: *full stack software engineer // building stuff at @AxiomZenTeam // adaptive like a chameleon // loves surfing // stoked on life*
- ***Ryuichi Okumura***

  @okuryu: *Technical @ Yahoo! JAPAN, made in Kyoto.*
- ***Open Source Weekends***

  @os_weekends: *#Meetup para amantes del #OpenSource. Nos reunimos una mañana de sábado al mes | Organizan @Josetekh, @kom_256, @ignaciodenuevo y @CodingCarlos*
- ***Paul Irish***

  @paul_irish: *The web is awesome, let's make it even better • I work on Chrome DevTools and browser performance • big fan of rye whiskey, data and whimsy*
- ***Axel Rauschmayer***

  @rauschma: *JavaScript: blogger @2ality, trainer @Ecmanauten, books http://exploringjs.com, newsletter @ESnextNews.*
- ***rem***

  @rem: *All about CSS sizing units. Sometimes about code too.*
- ***Brian Rinaldi***

  @remotesynth: *I share links on web, mobile development & tech. Work @ Progress (Telerik). Co-edit Mobile Web Weekly http://mobilewebweekly.co/ . Opinions no longer expressed here*
- ***Rebecca Murphey***

  @rmurphey: *Software engineer @IndeedEng. Tweets are my own, obvi.*
- ***Kristian Roebuck***

  @roebuk: *Web Developer, Runner*
- ***rick waldron***

  @rwaldron: *Open Web Engineer at Bocoup. Ecma/TC39. http://github.com/rwaldron Creator of Johnny-Five: http://johnny-five.io Black Lives Matter*
- ***stephan lindauer***

  @stephanlindauer: *#devops at @civey_de // computer science master student at @HTW_-Berlin // opinions are someone else's // hier koennte ihre werbung stehen*
- ***Profunctor Optics***

  @tomdale: *JavaScript thinkfluencer*
- ***Trevor Norris***

  @trevnorris: *node.js core maintainer and destroyer of slow code @NodeSource*
- ***Umar Hansa***

  @umaar: *Web developer.*

- ***WeCodeSign Podcast***

  @wecodesign: *El #podcast de diseño, desarrollo front-end y ux quincenal por @Ignacio-deNuevo - Feed: http://feeds.feedburner.com/WecodesignPodcast …*
- ***Yotam Ofek***

  @yotamofek: *http://snd.sc/yUPbtT*